Oracle для профессионалов

Том Кайт

торгово-издательскийдом
DiaSoft
Москва • Санкт-Петербург • Киев
2003

УДК **681.3.** 06(075)
ББК 32.973.2

К 91
КАЙТ ТОМ

К 91 Oracle для профессионалов. Пер. с англ./ТомКайт- СПб.: ООО «ДиаСофтЮП», 2003. — 672 с.

ISBN 5-93772-072-5

Выход в свет в конце прошлого года этой книги издательства Wrox стал эпохальным событием: впервые доходчиво и исчерпывающе раскрыты основные особенности архитектуры СУБД Oracle, принципиально отличающие ее от других популярных систем управления базами данных. Причем подробно описаны и проиллюстрированы множеством примеров именно те возможности, средства и особенности Oracle, которые обеспечивают разработку эффективных приложений для этой СУБД и ее успешную эксплуатацию.

Автор книги, Том Кайт, давно работает с СУБД Oracle, создает приложения и администрирует базы данных. Многие годы он профессионально занимается решением проблем, возникающих при использовании СУБД Oracle у администраторов и разработчиков по всему миру. На специализированном сайте корпорации Oracle **(http://asktom.oracle.com)** Том Кайт отвечает на десятки вопросов в день. Он не только делится знаниями, но и умело подталкивает читателя к самостоятельным экспериментам. Следуя по указанному им пути, становишься Профессионалом.

Если вы приступаете к изучению СУБД Oracle, — начните с этой книги. Если вы опытный разработчик приложений или администратор баз данных Oracle, — прочтите ее и проверьте, достаточно ли глубоко вы знаете эту СУБД и умеете ли использовать ее возможности. Вы найдете в книге десятки советов, описаний приемов и методов решения задач, о которых никогда не подозревали.

ББК 32.973.2

ISBN 5-93772-072-5 (рус.)
ISBN 1-861004-82-6 (англ.)

Гигиеническое заключение № 77.99.6.953.П.438.2.99 от 04.02.1999

Предисловие научного редактора

Уважаемые читатели!

О СУБД Oracle написано и издано как на английском, так и на русском языке, наверное, больше книг, чем обо всех остальных реляционных СУБД вместе взятых. Многие из этих книг полезны и интересны. Кроме того, в сети Internet доступны десятки тысяч страниц документации, которой могут воспользоваться разработчики, администраторы баз данных Oracle и пользователи. В журналах, на корпоративном сайте Oracle **(http://www.oracle.com)** и на десятках независимых сайтов печатаются статьи по всем аспектам администрирования и использования этой СУБД.

Тем не менее появление в конце прошлого года именно этой книги издательства Wrox стало эпохальным событием: впервые доходчиво и исчерпывающе раскрыты основные особенности архитектуры СУБД Oracle, принципиально отличающие ее от других популярных систем управления базами данных. Причем подробно описаны и проиллюстрированы множеством примеров именно те возможности, средства и особенности Oracle, которые обеспечивают разработку эффективных приложений для этой СУБД и ее успешную эксплуатацию.

В этом нет ничего удивительного. Автор книги, Том Кайт (Tom Kyte), давно работает с СУБД Oracle, создает приложения и администрирует несколько баз данных. Многие годы он, сотрудник корпорации Oracle, профессионально занимается решением проблем, возникающих при использовании СУБД Oracle у администраторов и разработчиков по всему миру. В дискуссионных группах **comp.databases.oracle.*** и в журнале Oracle Magazine, а в последние годы — еще и на специализированном сайте корпорации Oracle **(http://asktom.oracle.com)** Том Кайт отвечает на десятки вопросов в день.

Я набрел на этот сайт случайно, по ссылке в одном из документов, найденном славной поисковой системой Google. Потом стал его еженедельным, а со временем - ежедневным посетителем. Меня привлекло многообразие рассматриваемых проблем и изящество их решения. В изложении Тома Кайта все становится простым. При этом он не только делится знаниями, но и умело подталкивает читателя к самостоятельным экспериментам. Следуя по указанному им пути, становишься Профессионалом.

Поэтому меня очень обрадовало сообщение о выходе книги Тома Кайта «Expert one-on-one: Oracle», и я сделал все от меня зависящее, чтобы издательство «ДиаСофт» опубликовало ее перевод. Этот перевод я не смог никому доверить и сделал его сам. Если текст на русском языке понятен и приятен для чтения, то только благодаря потрясающему литературному редактору Ж. Е. Прусаковой, помогавшей мне в работе над книгой. Я благодарен ей, а также сотрудникам издательства «ДиаСофт» за поддержку и усилия, которые они приложили для выхода этой книги в свет.

В оригинале книга — очень большая, более 1200 страниц. В русском переводе, с разрешения издательства Wrox, она разделена на две части. Мы хотели часть книги, наиболее принципиальную для успешного использования СУБД Oracle, выпустить как можно быстрее. Сейчас, когда вы читаете первую часть, посвященную архитектуре и основным возможностям СУБД Oracle, продолжается работа над второй частью; она выйдет несколькими месяцами позже.

Мы старались сделать книгу максимально полезной и удобной для чтения: исправили выявленные читателями английской версии ошибки, которые приведены на сайте издательства Wrox, согласовали терминологию.

Если вы приступаете к изучению СУБД Oracle, — начните с этой книги. Если вы опытный разработчик приложений или администратор баз данных Oracle, — прочтите ее и проверьте, достаточно ли глубоко вы знаете эту СУБД и умеете ли использовать ее возможности. Вы найдете в книге десятки советов, описаний приемов и методов решения задач, о которых никогда не подозревали.

И, конечно же, рекомендую читать блестящие ответы и рекомендации Тома Кайта на его ежедневно обновляющемся сайте **AskTom: http://asktom.oracle.com.** Если корпорация Oracle станет общепризнанным и неоспоримым лидером на рынке производителей программного обеспечения, то в немалой степени благодаря тому, **что** и **как** делает для пользователей ее программных продуктов один человек — Том Кайт!

В. Кравчук,
OpenXS Initiative **(http://openxs.net)**
12 декабря 2002 года

Оглавление

Об авторе

Меня зовут Том Кайт. Я работаю в корпорации Oracle со времени версии 7.0.9 (для тех, кто не измеряет время версиями Oracle, уточню — с 1993 года). Однако я работал с СУБД Oracle, начиная с версии 5.1.5c (однопользовательская версия для DOS стоимостью 99 долларов на дискетах емкостью 360 Кбайт). До корпорации Oracle я более шести лет работал системным интегратором, занимаясь построением крупномасштабных гетерогенных баз данных и приложений (в основном для правительственных и оборонных учреждений). В этот период я много работал с СУБД Oracle, а точнее, помогал пользователям баз данных Oracle. Я работал непосредственно с клиентами на этапе создания спецификаций и построения систем, но чаще помогал перестраивать или настраивать системы ("настройка" обычно сводилась к переделке). Кроме того, я — именно тот Том, который ведет рубрику "AskTom" в журнале Oracle Magazine, отвечая на вопросы читателей о сервере и инструментальных средствах Oracle. Обычно на сайте **http://asktom.oracle.com** я получаю за день от 30 до 40 вопросов и отвечаю на них. Раз в два месяца я публикую подборку лучших вопросов с ответами в журнале (ответы на все вопросы доступны в Web и, естественно, хранятся в базе данных Oracle). В общем, я занимаюсь тем, что помогаю людям успешно эксплуатировать СУБД Oracle. Да, а в свободное время я разрабатываю приложения и программное обеспечение в самой корпорации Oracle.

В этой книге описано то, чем я занимаюсь ежедневно. Предлагаемый в ней материал посвящен темам и вопросам, с которыми пользователи сталкиваются постоянно. Все проблемы рассматриваются с позиции "если *я* использую это, то делаю *так...*". Моя книга — итог многолетнего опыта использования СУБД Oracle в тысячах различных ситуаций.

Благодарности

Я хотел бы поблагодарить многих людей, помогавших мне создать эту книгу. В корпорации Oracle я работаю с лучшими и наиболее яркими людьми из тех, кого мне удалось узнать, и они все так или иначе помогли мне. В частности, я благодарю Джоэла Калмана (Joel Kallman) за помощь в создании раздела книги, посвященного технологии interMedia. В ходе работы над сайтом AskTom мне не раз пришлось обращаться к Джоэлу за помощью в этой области — он именно тот человек, к которому стоит обратиться, если речь идет об interMedia и соответствующих технологиях. Я также благодарен Дэвиду Ноксу (David Knox) за помощь в создании примеров работы с протоколом SSL в разделе, посвященном пакету UTL_HTTP. Если бы не его знания и желание поделиться ими со мной, этого раздела просто не было бы. Наконец, я хочу поблагодарить всех, с кем работаю, за поддержку в испытании писательским трудом. Мне понадобилось намного больше времени и энергии, чем я мог себе представить, и я благодарен им за понимание моих проблем. В особенности, хочу поблагодарить Тима Хёхста (Tim Hoechst) и Майка Хичва (Mike Hichwa), которых я знаю по совместной работе уже почти 10 лет. Их постоянные вопросы и требования помогли мне разобраться с вещами, которыми я лично никогда и не подумал бы заниматься.

Хочу также поблагодарить пользователей программного обеспечения Oracle, задающих так много хороших вопросов. Если бы не они, мне и в голову не пришло бы написать эту книгу. Большая часть представленной здесь информации является прямым результатом чьего-нибудь вопроса "как" или "почему".

Наконец, и это самое главное, я благодарен за неизменную поддержку моей семье. Когда в тысячный раз слышишь: "Папа, ну почему ты все еще пишешь эту книгу?", то понимаешь, что кому-то нужен. Я просто не представляю, как бы я закончил эту книгу без постоянной поддержки моей жены Лори, сына Алана и дочери Мэган.

Введение

Представленный в этой книге материал сформирован на основе моего опыта разработки приложений Oracle и работы с коллегами-программистами, которым я помогал создавать надежные и устойчивые приложения для СУБД Oracle. Книга — лишь отражение того, чем я занимаюсь ежедневно, и тех проблем, с которыми постоянно сталкиваются люди.

Я описал то, что мне показалось наиболее важным, — базу данных Oracle и ее архитектуру. Я мог бы написать книгу с аналогичным названием, объясняющую, как разрабатывать приложения с помощью конкретного языка и архитектуры, например, с помощью технологии динамических HTML-страниц Java Server Pages, обращающихся к компонентам Enterprise Java Beans, которые используют интерфейс JDBC для взаимодействия с сервером Oracle. Однако в конечном итоге для успешного создания подобного приложения действительно необходимо понимать темы, представленные в этой книге. Книга посвящена тому, что, по моему мнению, должно быть известно всем для успешной разработки приложений Oracle, независимо от того, программируются ли эти приложения на Visual Basic с использованием интерфейса ODBC, на языке Java с использованием технологии EJB и интерфейса JDBC или на языке Perl с помощью модуля DBI. Эта книга не навязывает определенную архитектуру приложений; я не сравниваю трехуровневую архитектуру и архитектуру клиент-сервер. Здесь описано, что может делать база данных и что необходимо знать об особенностях ее работы. Поскольку база данных является основой любой архитектуры приложений, книга предназначена для широкой аудитории разработчиков.

О чем эта книга

Одна из проблем при наличии множества вариантов разработки — выяснить, какой из них лучше всего подходит в конкретной ситуации. Все хотят получить максимальную гибкость (как можно больше вариантов), но при этом так, чтобы все было просто и понятно. Oracle дает разработчикам практически неограниченный выбор. Никто еще не говорил: "Этого нельзя сделать в Oracle"; говорят иначе: "Сколько способов сделать это в Oracle вам надо?". Я надеюсь, что книга поможет вам сделать правильный выбор.

Эта книга для тех, кто приветствует свободу выбора, но хотел бы получить рекомендации и узнать особенности реализации тех или иных средств и функций Oracle. Например, СУБД Oracle предлагает замечательную возможность создания *виртуальной приватной базы данных* (virtual private database). В документации Oracle описано, как использовать эту возможность и что она дает. В документации, однако, не сказано, **когда** ее использовать и, что видимо еще важнее, когда ее не надо использовать. В документации не всегда представлены детали реализации той или иной возможности, а если они не известны, то могут неожиданно встать на вашем пути. (Речь идет не об ошибках, но о предполагавшихся способах работы и первоначальном назначении соответствующих средств.)

Для кого предназначена эта книга?

Целевой аудиторией являются все разработчики приложений для Oracle как сервера баз данных. Книга написана для профессиональных разработчиков Oracle, интересующихся тем, как решить задачу средствами этой СУБД. Практическая ориентация книги означает, что многие разделы будут очень интересны администраторам баз данных (АБД). В большинстве примеров, приведенных в книге для демонстрации ключевых возможностей, используется утилита SQL*Plus, поэтому вы не сможете узнать из нее, как разрабатывать удобный и красивый графический пользовательский интерфейс, но зато узнаете, как работает СУБД Oracle, что позволяют сделать ее основные средства и когда их нужно (и не нужно) использовать.

Книга предназначена для тех, кто хочет получить от СУБД Oracle большую отдачу при меньших усилиях. Она для каждого, кто хочет знать, как средства Oracle могут применяться в практической работе (при этом не только приводятся примеры использования того или иного средства, но и объясняется, почему это средство необходимо). Еще одна категория людей, которым эта книга покажется интересной, — технические руководители групп разработчиков, реализующих проекты на базе Oracle. В некотором отношении очень важно, чтобы они знали особенности СУБД, имеющие принципиальное значение для успешной разработки. Эта книга может дать дополнительные аргументы руководителю проекта, желающему обучить персонал необходимым технологиям или убедиться, что разработчики уже знают то, что им необходимо знать.

Чтобы получить максимальную отдачу от этой книги, читатель должен:

О Знать **язык SQL.** Не обязательно быть лучшим специалистом по SQL в стране, но хорошее практическое знание существенно поможет.

- **Понимать программы на языке PL/SQL.** Это не обязательное требование, но знание PL/SQL поможет "воспринять" примеры. Эта книга, например, не объясняет, как создавать циклы **FOR** или объявлять тип записи — об этом достаточно написано в документации Oracle и в многочисленных книгах. Однако это не значит, что вы не узнаете много нового о программировании на PL/SQL, прочтя эту книгу. Узнаете. Вы очень близко познакомитесь со многими возможностями PL/SQL и узнаете новые способы решения задач, изучите пакеты и средства, о существовании которых, возможно, даже и не подозревали.
- **Иметь определенный опыт работы с процедурным языком программирования, например C или Java.** Я уверен, что любой, кто способен понимать и писать код на каком-то процедурном языке программирования, сможет успешно разобраться в примерах, представленных в книге.
- **Ознакомиться с руководством Oracle Server Concepts Manual.**

Несколько слов об этом руководстве. Из-за большого объема многих пугает документация Oracle. Если вы только начинаете изучение руководства или ранее не читали подобной документации, я рекомендую начать именно с *Oracle8i Concepts.* Его объем — около 800 страниц, и в нем описаны многие из фундаментальных концепций Oracle, о которых вам надо знать. Это руководство не описывает все технические детали (именно этому посвящены остальные 10-20 тысяч страниц документации), но представляет все важнейшие концепции. В руководстве затронуты, в частности, следующие темы:

- структура базы данных, организация и хранение данных;
- распределенная обработка;
- архитектура памяти Oracle;
- архитектура процессов Oracle;
- объекты пользовательской схемы (таблицы, индексы, кластеры и т.д.);
- встроенные и определяемые пользователем типы данных;
- хранимые процедуры;
- особенности организации транзакций;
- оптимизатор;
- целостность данных;
- управление одновременным доступом.

Я и сам периодически перечитываю соответствующие главы. Это — фундаментальные концепции. Если вы их не понимаете, создаваемые вами приложения Oracle будут обречены на провал. Я рекомендую прочитать это руководство и разобраться хотя бы в важнейших вещах.

Структура книги

Чтобы облегчить восприятие, книга поделена на шесть отдельных частей (они описаны ниже). Это не просто структурные единицы — они помогут быстрее найти наиболее существенную информацию. Книга состоит из 23 глав, каждая из которых — "мини-книга", то есть практически отдельный компонент. Изредка я ссылаюсь на примеры или возможности, описанные в других главах (часть, посвященная защите, например, больше других зависит от примеров и понятий, представленных в других главах). Но, как правило, вполне можно читать главу отдельно от остальной книги. Например, вовсе не нужно прочитать главу 10, чтобы понять главу 14.

Форматы и стили глав практически идентичны:

- Введение в описываемые средства или возможности.
- Почему это нужно (или не нужно) использовать. Я описываю ситуации, когда имеет смысл использовать данное средство и когда этого делать не стоит.
- Как это использовать. Это не просто цитата из справочного руководства по языку SQL, а пошаговое описание: вот что требуется, что для этого надо сделать, а вот предварительные условия применения. В этом разделе рассматривается:

  Как применить то или иное средство или прием

  Примеры, примеры и еще примеры

  Отладка, поиск ошибок при реализации

  Подводные камни при использовании средства

  Устранение ошибок (превентивное)

- Резюме, подводящее итог рассмотрения.

В книге содержится множество примеров и большое количество кода, причем весь этот код доступен для загрузки на сайте **http://www.wrox.com.** Далее представлено подробное содержание каждой части.

Основные понятия

- **Глава 1. Разработка успешных приложений для Oracle. В** этой главе описан мой принципиальный подход к программированию баз данных. Все СУБД устроены **по-разному,** и чтобы успешно и в срок разработать приложение, использующее базу данных, необходимо точно знать, **что** и **как** позволяет сделать ваша СУБД. Не зная возможностей СУБД, вы рискуете в очередной раз "изобрести велосипед", то есть создать средства, уже предоставляемые базой данных. Если вы не знаете, как работает СУБД, то с большой вероятностью будете создавать неэффективные и непредсказуемые приложения.

  В этой главе описывается ряд реальных приложений, при создании которых недостаток понимания базовых концепций СУБД привел к неудаче всего проекта. На основе такого практического подхода с контрпримерами в главе описываются базовые возможности и функции СУБД, которые необходимо понимать

разработчикам. Идея в том, что вы не можете себе позволить рассматривать СУБД как черный ящик, просто "отвечающий на запросы" и самостоятельно обеспечивающий требуемую масштабируемость и производительность.

- Глава 2. Архитектура. СУБД Oracle — весьма сложная система. При каждом подключении к базе данных или выполнении команды **UPDATE** в фоновом режиме работает целый набор процессов, гарантирующих устойчивую работу приложений и поддержку целостности данных. Например, СУБД поддерживает достаточный объем информации для восстановления данных в исходное состояние при необходимости. Она кэширует данные и в случае необходимости повторно использует их. И так далее. В большинстве случаев все это происходит незаметно (по крайней мере, для разработчика), но при возникновении проблем половина успеха зависит от знания того, где искать их причину.

  В этой главе описаны три основных компонента архитектуры Oracle: структура памяти (в частности, глобальная системная область — System Global Area), физические процессы и набор файлов (фалы параметров, файлы журнала повторного выполнения...). Знание архитектуры Oracle принципиально важно для понимания уникального способа реализации ряда возможностей в Oracle и его влияния на приложения.

- **Глава 3. Блокирование и одновременный доступ.** Различные СУБД выполняют одни и те же операции по-разному (то, что хорошо работает в SQL Server, может гораздо хуже работать в Oracle), и понимание реализации механизмов блокирования и управления одновременным доступом в Oracle жизненно важно для создания успешно работающих приложений.

  В этой главе описан базовый подход к этим механизмам, используемый в СУБД Oracle, типы применяемых блокировок (ЯМД, ЯОД, защелки...) и проблемы, возникающие при некорректной реализации блокирования (взаимные блокировки, блокирование-доступа и эскалация блокировок). В разделе, посвященном управлению одновременным доступом, описываются предоставляемые СУБД Oracle функции, которые позволяют контролировать доступ пользователей к базе данных и ее изменение.

- **Глава 4. Транзакции.** Транзакции — фундаментальное свойство всех баз данных; именно они отчасти отличают базу данных от файловой системы. И тем не менее их часто понимают неправильно, а многие разработчики даже не подозревают, что вовсе их не используют. В этой главе описано, как правильно использовать транзакции в СУБД Oracle. Кроме того, автор обращает внимание на ряд "плохих привычек", приобретаемых при разработке приложений для других СУБД. В частности, рассматривается понятие неделимости транзакции и ее влияние на выполнение операторов в Oracle. Затем описываются операторы управления транзакциями **(COMMIT, SAVEPOINT, ROLLBACK),** ограничения целостности и распределенные транзакции (протокол двухфазной фиксации). Наконец, рассматривается ряд реальных проблем использования транзакций: их регистрация и роль журнала повторного выполнения и сегментов отката.

Структуры и утилиты базы данных

- **Глава 5. Повторное выполнение и откат.** Можно сказать, что разработчику не требуется столь же детально, как АБД, понимать работу журнала повторного выполнения и сегментов отката, но разработчик должен представлять их значение для базы данных. Обсудив назначение журнала повторного выполнения, мы опишем, что конкретно делает оператор **COMMIT.** Мы также рассмотрим ряд проблем, в частности объем генерируемой информации в журнале повторного выполнения, отключение журнализации, а также проанализируем использование сегментов отката.

  В разделе главы, посвященном откату, мы сначала рассмотрим, какие операторы генерируют наибольший и наименьший объем данных для отката, а затем перейдем к SQL-оператору **set transaction.** Он обычно используется для задания большого сегмента отката для операции с большим объемом изменений. Затем мы займемся печально знаменитой ошибкой **'ORA-01555 snapshot too old',** рассмотрим причины ее возникновения и способы предотвращения.

- **Глава 6. Таблицы.** Oracle сейчас поддерживает несколько типов таблиц. В этой главе рассмотрены все типы таблиц — произвольно организованные (обычные, "нормальные" таблицы), организованные по индексу, входящие в индексный кластер, входящие в хеш-кластер, вложенные, временные и объектные — и описано, когда, как и почему имеет смысл использовать тот или иной тип. В большинстве случаев произвольно организованных таблиц вполне достаточно, но вы должны понимать, когда другие типы могут оказаться более подходящими.

- **Глава 7. Индексы.** Индексы — критический компонент проекта приложения. Правильная реализация индексов требует глубокого знания данных, их распределения и способов их использования. Слишком часто индексы считают второстепенными структурами при разработке приложений, и от этого снижается их производительность.

  В этой главе мы детально рассмотрим различные типы индексов, включая индексы на основе B-дерева, битовые индексы, индексы по функциям и индексы прикладных доменов, а также опишем, когда надо и не надо использовать индексы определенного типа. В разделе "Часто задаваемые вопросы" мы дадим ответы на некоторые из типичных вопросов типа "Работают ли индексы с представлениями?" и "А почему мой индекс не используется?".

- **Глава 8. Импорт и экспорт.** Средства импорта и экспорта — одни из самых старых инструментальных средств Oracle, используемые для извлечения таблиц, схем или всего определения базы данных из одного экземпляра Oracle и переноса их на другой экземпляр или в другую схему. Тем не менее, многие разработчики не знают, как правильно их использовать. Мы рассматриваем проблемы экспорта больших объемов информации, разделения и переноса данных, и использования этих средств при резервном копировании или реорганизации данных. Глава заканчивается описанием потенциальных ловушек и проблем при использовании средств экспорта и импорта данных.

- **Глава 9. Загрузка данных.** Эта глава посвящена утилите SQLLDR и описывает различные способы использования этого средства для загрузки и изменения данных в базе данных. Среди рассмотренных тем — загрузка данных в формате записей с разделителями полей, обновление существующих строк и вставка новых, выгрузка данных и вызов утилиты SQLLDR из хранимой процедуры. Хотя SQLLDR тоже является давним и принципиально важным средством, но в отношении его практического использования возникает много вопросов.

Производительность

- **Глава 10. Стратегии и средства настройки.** Это одна из моих любимых тем, и в данной главе я детально описываю свой подход к настройке производительности приложений Oracle, а затем предлагаю удобное практическое руководство по применяемым средствам и методам. Начальный раздел посвящен настройке приложений, включая такие темы, как связываемые переменные и синтаксический анализ, утилиты **SQL_TRACE, TIMED_STATISTICS** и **TKPROF,** пакет **DBMS_PROFILER** и важность наличия средств журнализации в приложениях. После полной настройки приложений можно переходить к базе данных, в частности к группе утилит **StatsPack** и представлениям V$, широко используемым при настройке.

- **Глава 11. Стабилизация плана оптимизатора.** Разработчики, использующие Oracle 8i (и более новые версии), теперь могут сохранять набор "подсказок серверу", известный как план оптимизатора и детально описывающий, как лучше выполнять в базе данных определенный SQL-оператор. Это дает несомненные преимущества с точки зрения производительности, и мы детально описываем способ генерации таких "шаблонов" (outlines) и управление ими.

Расширенные возможности SQL

- **Глава 12. Аналитические функции.** Некоторые вопросы об информации в базе данных задают часто, но реализующие их запросы сложно написать на обычном языке SQL (кроме того, такие запросы не всегда быстро работают). Сервер Oracle 8.1.6 впервые предоставил аналитические функции. Эти функции добавляют расширения языка SQL, упрощающие создание такого рода запросов и существенно повышающие производительность по сравнению с обычным SQL-запросом. В этой главе описан принцип работы аналитических функций, рассматривается полный синтаксис (включая конструкции функции, фрагмента и окна), а затем даются полные практические примеры использования этих функций.

- **Глава 13. Материализованные представления.** Некоторые "агрегирующие" запросы должны для получения ответа обрабатывать терабайты данных. Последствия с точки зрения производительности очевидны, особенно если речь идет о часто выполняемом запросе — необходимость обработки огромного объема данных при каждом его выполнении. При использовании материализованных представлений мы просто делаем основную часть работы заранее — собираем

данные, необходимые для ответа на запрос в материализованном представлении, и дальнейшие запросы выполняются к этим заранее собранным данным. Более того, СУБД может распознавать аналогичные запросы, использующие эти подготовленные данные, и автоматически переписывать запрос так, чтобы он использовал материализованное представление. В этой главе описано, как все это работает и как поддерживать материализованные представления, включая использование ограничений, измерений и пакета **DBMS_OLAP,**

- **Глава 14. Фрагментация.** Фрагментация создавалась как средство для упрощения управления очень большими таблицами и индексами за счет использования подхода "разделяй и властвуй" — деления таблицы или индекса на несколько меньших и более управляемых частей. Это как раз та область, где АБД и разработчик должны работать вместе с целью обеспечения максимальной доступности и производительности приложения. В данной главе рассмотрена фрагментация как таблиц, так и индексов. Мы рассмотрим фрагментацию с использованием локальных индексов (типична для хранилищ данных) и глобальных индексов (обычно используется в системах оперативной обработки транзакций — ООТ).

- **Глава 15. Автономные транзакции.** Это средство позволяет создать подтранзакцию, изменения которой фиксируются или откатываются независимо от родительской транзакции. Мы рассмотрим ситуации, когда это может понадобиться, например отслеживание "нелегальных" попыток изменить защищенную информацию, попытки избежать ошибки "изменяющейся таблицы" или выполнение операторов ЯОД в триггерах. Затрагиваются также проблемы управления транзакциями, области действия, завершения автономных транзакций и точек сохранения.

- **Глава 16. Динамический SQL.** В этой главе сравниваются два метода использования SQL-операторов в программах: "обычный" статический SQL и динамический SQL. Динамический SQL-код — это SQL-код, формируемый по ходу выполнения, не известный на этапе компиляции. Мы рассмотрим два метода использования динамического SQL в программах: на основе стандартного пакета DBMS_SQL и встроенного динамического SQL, декларативного метода для программ на языке PL/SQL. В каждом конкретном случае может быть несколько причин выбрать один из этих методов, например известность связываемых переменных на этапе компиляции, известность вида результатов на этапе компиляции, одно- или многократное выполнение динамически формируемого оператора в ходе сеанса. Мы детально рассмотрим все эти причины.

Расширяемость

- **Глава 17. interMedia.** Эта глава посвящена компоненту interMedia Text. Вместо подробного описания того, "как использовать interMedia Text", мы рассмотрим, что это такое и что этот компонент может дать, а также средства СУБД, позволяющие достичь соответствующей функциональности. Мы рассмотрим поиск текста, возможности управлять разнообразными документами, индекси-

ровать текст из различных источников данных и искать приложения XML. Глава завершается описанием ряда проблем interMedia, в том числе синхронизации индексов и индексирования информации, хранящейся вне базы данных.

- **Глава 18. Внешние процедуры на языке С.** Начиная с Oracle 8.0, появилась возможность подключать к СУБД процедуры, реализованные на языках, отличных от PL/SQL, например, С или Java. Такие процедуры называют **внешними. В** этой главе мы рассмотрим процедуры на языке С с точки зрения архитектуры. Мы опишем, как сконфигурировать сервер для использования таких процедур, как протестировать конфигурацию, и создадим пример процедуры, передающей и обрабатывающей переменные различных типов. Мы также рассмотрим внешнюю процедуру для преобразования большого двоичного объекта в файл **(LOB_IO),** позволяющую записывать на диск большие двоичные объекты типа **CLOB, BLOB** и **BF1LE.**

- **Глава 19. Хранимые процедуры на языке Java.** За счет разумного использования небольших фрагментов Java-кода можно получить широкие функциональные возможности, недоступные в языке PL/SQL. В этой главе рассматриваются практические примеры правильного использования этой возможности, например, для получения списка файлов в каталоге или выполнения команды операционной системы. Глава заканчивается описанием ряда проблем, с которыми может столкнуться разработчик при попытке использовать это средство, и возможных решений этих проблем.

- **Глава 20. Использование объектно-реляционных средств.** Наличие объектно-реляционных возможностей в СУБД (начиная с Oracle 8i) существенно расширяет набор доступных разработчику типов данных. Но когда их использовать (и, что не менее важно, стоит ли использовать)? В этой главе показано, как добавлять в систему новые типы данных (мы создадим новый тип данных PL/SQL), и рассмотрены уникальные возможности, предоставляемые наборами. Наконец, мы рассмотрим объектно-реляционные представления, предназначенные для тех, кто хочет воспользоваться объектно-реляционными возможностями, не теряя реляционного представления данных приложения.

Защита

- **Глава 21. Тщательный контроль доступа.** Это средство позволяет добавить условие ко всем выполняемым запросам к базе данных. Реализация такой возможности на сервере означает, что ею может воспользоваться любое приложение, обращающееся к базе данных. Дополнительные аргументы в пользу применения этого средства — простота сопровождения и возможность устанавливать приложение на сервере приложений. Мы рассмотрим, как работает контроль доступа, путем тестирования пары примеров, реализующих на его основе правила защиты и использующих контексты приложений. Глава завершается разделом о потенциальных проблемах, в том числе при обеспечении целостности ссылок, экспорте и импорте данных, а также описанием возможных ошибок.

- **Глава 22. Многоуровневая аутентификация. В** этой главе рассматривается влияние среды Web, существенно увеличивающей количество ситуаций, когда клиенты регистрируются на промежуточном сервере приложений, прежде чем получить доступ к базе данных. Мы рассмотрим, как реализовать такую возможность и что при этом происходит. Мы также рассмотрим, как предоставить привилегии и организовать проверку действий промежуточных учетных записей.
- **Глава 23. Права вызывающего и создателя.** Начиная с Oracle 8i можно предоставлять различные наборы привилегий отдельным пользователям хранимой процедуры. При установке прав вызывающего можно создавать хранимые процедуры, выполняемые с набором привилегий вызывающего пользователя. Мы рассмотрим, для чего может пригодиться такая возможность, в частности, при создании утилит общего назначения и приложений словаря данных, и почему в большинстве случаев правильным остается выполнение с правами создателя. В разделе "как это работает" мы рассмотрим детально, что происходит при компиляции процедур с правами создателя и вызывающего.

Приложения

- **Приложение А. Основные стандартные пакеты.** Многие из этих пакетов при разработке не используются, или их назначение неверно интерпретируется. В этом приложении я пытаюсь объяснить их назначение, показать, как их использовать и расширять.

Типографские особенности книги

Мы используем различные стили оформления текста и типографские соглашения, помогающие выделить различные виды информации. Вот примеры используемых стилей с объяснением их назначения.

Код выделяется по-разному. Если это слово, описываемое в тексте (например, если речь идет об операторе **SELECT),** фрагмент кода выделяется полужирным. Если это блок кода, набираемый и выполняемый, например, в SQL*Plus, он представлен специальным шрифтом:

```
tkyte@DEV816> create or replace procedure StaticEmpProc(p_job in varchar2)
  2  as
  3     begin
  4       for x in  (select ename from emp where job = p_job)
  5          loop
  6             dbms_output.put_line(x.ename);
  7          end loop;
  8   end;
  9   /

Procedure created.
```

Мы также показываем номера строк, выдаваемые при вводе в сеансе SQL*PIus, — это упрощает ссылки на них.

Советы, подсказки и дополнительная информация представлены в таком стиле.

Важная информация представлена в таких блоках.

Термины выделены наклонным шрифтом.

Маркированные списки представлены со сдвигом вправо. В этих списках:

- **важные слова** выделены полужирным шрифтом;
- слова, которые можно увидеть в меню на экране, например **File** или **Window,** выделены полужирным шрифтом;
- клавиши, которые необходимо нажимать на клавиатуре, например Ctrl или *Enter,* выделены наклонным шрифтом.

Контакты с читателями

Мы попытались сделать эту книгу максимально точной и удобной для изучения, но важно лишь, насколько она окажется полезной вам. Пожалуйста, поделитесь своим мнением о книге, либо послав нам карточку обратной связи, предлагаемую в конце книги, либо просто написав письмо по электронной почте по адресу **feedback@wrox.com.**

Исходный код и обновления

При работе с представленными в книге примерами вы можете предпочесть набирать весь код вручную. Многие читатели предпочитают именно это способ, поскольку так проще привыкнуть к используемым приемам кодирования.

Хотите вы набирать код или нет, мы предоставили все исходные коды для этой книги на сайте издательства по адресу:

http://www.wrox.com/

Если вы — один из тех, кто любит набирать код вручную, можете использовать эти файлы для сравнения с требуемыми результатами. Обратитесь к ним, если предполагаете, что при наборе сделали ошибку. Если же вы не любите набирать тексты, обязательно загрузите их с Web-сайта!

В любом случае исходные коды помогут при отладке примеров.

Ошибки

Мы сделали все возможное, чтобы избежать ошибок в тексте и исходных кодах. Однако людям свойственно ошибаться, и поэтому мы считаем своим долгом информировать читателей о любых обнаруженных и исправленных ошибках. Информацию об ошибках, обнаруженных во всех наших книгах, можно найти на сайте **http:// www.wrox.cora.** Если вы найдете ошибку, о которой еще не известно, пожалуйста, дайте нам знать.

Наш сайт является центром накопления справочной и другой технической информации, включая код из всех изданных книг, примеры глав, обзоры готовящихся к выходу книг, а также статьи и мнения по соответствующим темам.

Настройка среды

В этом разделе я опишу, как настроить среду для выполнения примеров из данной книги. Я опишу:

- как установить демонстрационную схему SCOTT/TIGER;
- среду, которую необходимо создать и запустить;
- как сконфигурировать средство AUTOTRACE в SQL* Plus;
- как настроить компилятор языка C;
- соглашения по оформлению кода, принятые в этой книге.

Установка демонстрационной схемы SCOTT/TIGER

Схема SCOTT/TIGER может уже существовать в базе данных в нескольких экземплярах. Она обычно создается при типичной установке, хотя и не является обязательным компонентом базы данных. Демонстрационную схему SCOTT можно установить в любую учетную запись пользователя — имя SCOTT не имеет никакого тайного смысла. Таблицы EMP/DEPT можно при желании создать в собственной схеме.

Многие из примеров в этой книге зависят от таблиц в схеме SCOTT. Если вы хотите их проверить, эти таблицы необходимы. Если вы работаете в общей базе данных, имеет смысл установить собственную копию соответствующих таблиц в некоторой схеме, отличающейся от схемы SCOTT, чтобы избежать побочных эффектов, вызванных использованием и изменением тех же данных другими пользователями.

Чтобы создать демонстрационные таблицы схемы **SCOTT,** необходимо:

- перейти в каталог **[ORACLE_HOME]/sqlplus/demo;**
- запустить сценарий **demobld.sql** от имени любого пользователя.

Сценарий **demobld.sql** создаст и наполнит данными пять таблиц. При завершении он автоматически завершает работу утилиты SQL*Plus, так что не удивляйтесь, если окно после запуска этого сценария SQL*Plus исчезнет — так и должно быть.

Стандартные демонстрационные таблицы включают стандартные требования целостности ссылок. Некоторые из моих примеров предполагают наличие этих требований. После выполнения сценария **demobld.sql** рекомендуется также выполнить следующие операторы:

```
alter table emp add constraint emp_pk primary key(empno);
alter table dept add constraint dept_pk primary key(deptno);
alter table emp add constraint emp_fk_dept
                                  foreign key (deptno) references dept;
alter table emp add constraint emp_fk_emp foreign key(mgr) references emp;
```

Они завершат установку необходимой демонстрационной схемы. Если понадобится удалить эту схему, можно просто выполнить сценарий **[ORACLE_HOME]/sqlplus/demo/demodrop.sql.** Он удалит все пять таблиц и завершит работу сеанса SQL*Plus.

Среда SQL*Plus

Почти все примеры в этой книге предназначены для выполнения в среде SQL*Plus. Исключение представляют примеры на языке C, для которых, разумеется, необходим компилятор C, кроме сервера Oracle (см. раздел "Компиляторы языка C" далее). За исключением этого, SQL*Plus — единственная утилита, которую необходимо настроить и сконфигурировать. Утилита SQL*Plus имеет много полезных опций и команд, которые часто будут использоваться по ходу изложения. Например, почти все примеры в книге так или иначе используют пакет **DBMS_OUTPUT.** Чтобы этот пакет работал, необходимо выполнить следующую команду SQL*Plus:

```
SQL> set server output on
```

Я думаю, вам быстро надоест постоянно ее набирать. К счастью, утилита SQL*Plus позволяет создать файл login.**sql** — сценарий, выполняемый при каждом запуске сеанса SQL*Plus. Более того, можно установить значение переменной среды **SQLPATH,** задающее местонахождение этого сценария начального запуска.

Для всех примеров в этой книге я использую сценарий **login.sql** следующего вида:

```
define _editor=vi

set serveroutput on size 1000000

set trimspool on
set long 5000
set linesize 100
set pagesize 9999
column plan_plus_exp format a80
```

```
column global_name new_value gname
set termout off
select lower(user) || '8' ||
decode(global_name, 'ORACLE8.WORLD', '8.0', 'ORA8I.WORLD',
'8i', global_name) global_name from global_name;
set sqlprompt '&gname>
set termout on
```

В этом сценарии:

- DEFINE _EDITOR=vi устанавливается стандартный редактор для SQL*Plus. Можете задать свой любимый текстовый редактор (но не текстовый процессор), например Notepad или emacs.
- SET SERVEROUTPUT ON SIZE **1000000** включает по умолчанию поддержку пакета DBMS_OUTPUT (чтобы не нужно было набирать эту команду каждый раз). Устанавливает также максимально возможный размер буфера.
- SET **TRIMSPOOL** ON гарантирует, что в выдаваемом тексте хвостовые пробелы будут отсекаться. Если используется стандартное значение этой установки, **OFF,** выдаваемые строки будут иметь длину **LINESIZE.**
- SET LONG **5000** устанавливает стандартное количество байт, выдаваемых при выборе столбцов типа LONG и **CLOB.**
- SET **LINESIZE 100** устанавливает длину выдаваемой утилитой SQL*Plus строки равной 100 символам.
- SET **PAGESIZE 9999** задает значение **PAGESIZE,** определяющее, как часто утилита SQL*Plus выдает заголовки, настолько большим, чтобы на странице было не более одного заголовка.
- COLUMN PLAN_PLUS_EXP FORMAT **A80** устанавливает формат для результатов выполнения оператора **EXPLAIN PLAN,** получаемых при установке опции AUTOTRACE. Формат **A80** (ширина — 80 символов) обычно достаточен для представления большинства планов.

Следующая часть сценария **login.sql** задает приглашение SQL*Plus. Она начинается со строки:

```
column global_name new value gname
```

Эта директива заставляет утилиту SQL*Plus помещать последнее значение, извлеченное из любого столбца с именем **GLOBAL_NAME,** в переменную подстановки **GNAME.** Затем выполняется следующий запрос:

```
select lower(user) || '@' ||
decode(global_name, 'ORACLE8.WORLD', '8.0', 'ORA8I.WORLD',
'8i', global_name) global_name from global_name;
```

Он выбирает из базы данных значение **GLOBAL_NAME,** используя функцию DECODE для присваивания требуемых имен одному или нескольким обычно используемым экземплярам, а затем конкатенирует его с именем текущего пользователя. Наконец, мы отображаем эту информацию в приглашении SQL*Plus:

```
set sqlprompt '&gname>'
```

Поэтому приглашение будет иметь вид:

```
tkyte@TKYTE816>
```

Таким образом, я знаю, **кто** я и где я. Еще один очень полезный сценарий, который можно поместить в тот же каталог, что и **login.sql,** — это сценарий **connect.sql:**

```
set termout off
connect &1
Slogan
set termout on
```

Утилита SQL*Plus будет выполнять сценарий **login.sql** при начальном запуске. В общем случае он должен выполняться при каждом подключении. Я просто завел себе привычку использовать команду:

```
tkyte@TKYTE816> @connect scott/tiger,
```

а не просто **CONNECT SCOTT/TIGER. В** результате мое приглашение всегда устанавливается должным образом, как и другие установки, такие как **SERVEROUTPUT.**

Настройка AUTOTRACE в SQL*Plus

По ходу всей книги эта установка пригодится нам для контроля производительности выполняемых запросов и получения информации о плане выполнения запроса, выбранном оптимизатором SQL, и другой полезной статистической информации о ходе выполнения. Oracle предлагает средство **EXPLAIN PLAN,** которое при использовании команды **EXPLAIN PLAN** позволяет генерировать план выполнения запроса.

*Подробнее об интерпретации результатов выполнения **EXPLAIN PLAN** см. в руководстве "Oracte8i Designingand TuningforPerformance".*

Однако утилита SQL*Plus предлагает средство **AUTOTRACE,** позволяющее получать планы выполнения обрабатываемых запросов, а также информацию об используемых ресурсах, без выполнения команды **EXPLAIN PLAN.** Соответствующий отчет генерируется после успешного выполнения операторов **ЯМД** (т.е. **SELECT, DELETE, UPDATE** и **INSERT). В** этой книге это средство широко используется. Средство AUTOTRACE можно настроить несколькими способами. Я практикую следующую последовательность действий:

- перехожу в каталог **[ORACLE_HOME]/rdbms/admin;**
- регистрируюсь в SQL*Plus от имени **SYSTEM;**
- запускаю сценарий **@utlxplan;**
- выполняю оператор **CREATE PUBLIC SYNONYM PLAN_TABLE FOR PLAN_TABLE;**
- выполняю оператор **GRANT ALL ON PLAN_TABLE TO PUBLIC.**

Если хотите, можете заменить **GRANT ... TO PUBLIC** оператором **GRANT** для конкретного пользователя. Предоставляя привилегию роли **PUBLIC,** вы фактически разрешаете трассировать операторы в SQL*Pius любому пользователю. По-моему это непло-

хо — пользователи могут не устанавливать собственные таблицы планов. Альтернатива этому — запуск сценария **@UTLXPLAN** в каждой схеме, где необходимо использовать средство **AUTOTRACE.**

Следующий шаг — создание и предоставление всем роли **PLUSTRACE:**

- переходим в каталог **[ORACLE_HOME]/sqlplus/admin;**
- регистрируемся в SQL*Plus от имени SYS;
- запускаем сценарий **@plustrce;**
- выполняем оператор **GRANT PLUSTRACE TO PUBLIC.**

И в этом случае, если хотите, можете заменить **PUBLIC** в операторе **GRANT** именем конкретного пользователя.

Управление отчетом о плане выполнения

Управлять информацией, выдаваемой в отчете о плане выполнения, можно с помощью установки системной переменной **AUTOTRACE.**

SET AUTOTRACE OFF	Отчет **AUTOTRACE** не генерируется. Так происходит по умолчанию.
SET AUTOTRACE ON EXPLAIN	В отчете **AUTOTRACE** показывается только выбранный оптимизатором план.
SET AUTOTRACE ON STATISTICS	В отчете **AUTOTRACE** показывается только статистическая информация о выполнении оператора SQL
SET AUTOTRACE ON	В отчет **AUTOTRACE** включается как выбранный оптимизатором план, так и статистическая информация о выполнении оператора SQL.
SET AUTOTRACE TRACEONLY	Аналогично **SET AUTOTRACE ON,** но подавляет выдачу результатов выполнения запроса.

Интерпретация плана выполнения запроса

План выполнения отражает выбранный оптимизатором способ выполнения запроса. Каждая строка плана выполнения имеет порядковый номер. Утилита SQL*Plus также выдает номер строки родительской операции.

План выполнения состоит из четырех столбцов, выдаваемых в следующем порядке:

Имя столбца	*Описание*
ID_PLUS_EXP	Показывает порядковый номер шага выполнения
PARENT_ID_PLUS_EXP	Показывает для каждого шага родительский шаг. Этот столбец полезен в больших отчетах.
PLAN_PLUS_EXP	Показывает описание шага выполнения.
OBJECT_NODE_PLUS_EXP	Показывает использованные базы данных или серверы для параллельного запроса.

Формат столбцов можно изменять с помощью команды **COLUMN.** Например, чтобы отменить выдачу столбца **PARENT_ID_PLUS_EXP,** введите:

```
SQL> column parent_id_plus_exp noprint
```

Компиляторы языка C

Поддерживаемые сервером Oracle компиляторы языка C зависят от операционной системы. В Microsoft Windows я использую Microsoft Visual C/C++. Я использую только средства командной строки **(nmake** и **cl). Ни** в одном из примеров не использовалась графическая среда разработки. Однако их можно при желании проверять и в этой среде. Вам придется самостоятельно сконфигурировать и настроить соответствующие файлы **include** и подключить нужные библиотеки. Все файлы управления проектом **makefile,** содержащиеся в данной книге, — очень маленькие и простые, из них вполне очевидно, какие файлы **include** и библиотеки необходимы.

В среде Sun Solaris поддерживается компилятор языка C, входящий в состав пакета Sun SparcsWorks. И в этом случае я использовал для компиляции программ только средства командной строки, **make** и cc.

Оформление кода

Единственная особенность оформления кода, на которую я хочу обратить внимание читателей, — это именование переменных в коде PL/SQL. Например, рассмотрим следующее тело пакета:

```
create or replace package body my_pkg
as
    g_variable varchar2(25);
    procedure p(p_variable in varchar2)
    is
        l_variable varchar2(25);
    begin
        null;
    end;
end;
/
```

В этом примере используются три переменные: глобальная переменная пакета **G_VARIABLE,** формальный параметр процедуры, **P_VARIABLE,** и, наконец, локальная переменная, **L_VARIABLE. Я** именую переменные в соответствии с областями действия: все глобальные имеют префикс G_, параметры — префикс P_, а локальные переменные — префикс L_. Главная причина этого — необходимость отличать переменные PL/SQL от столбцов таблицы базы данных. Например, рассмотрим следующую процедуру:

```
create procedure p(ENAME in varchar2)
as
begin
    for x in (select * from emp where ename = ENAME) loop
        dbms_output.put_line(x.empno);
    end loop;
end;
```

Она всегда будет выдавать все строки в таблице **EMP.** В операторе SQL конструкция **ename = ENAME** интерпретируется, конечно же, как сравнение столбца с самим собой. Можно использовать сравнение **ename = P.ENAME,** то есть уточнить ссылку на переменную PL/SQL именем процедуры, но об этом легко забыть, что приведет к возникновению ошибок.

Я всегда именую переменные в соответствии с областью действия. В этом случае я могу легко отличить параметры от локальных и глобальных переменных, а также не путать имена переменных и столбцов таблицы.

Другие особенности

Каждая глава в этой книге самодостаточна. В начале каждой главы я удалял свою тестовую учетную запись и создавал ее заново. То есть, каждая глава начиналась с чистой схемы — без объектов. Если выполнять все примеры, с начала до конца главы, следует делать именно так. При запросах к словарю данных в поисках объектов, созданных в результате выполнения той или иной команды, вас могут сбить с толку объекты, оставшиеся от других примеров. Кроме того, я часто повторно использую имена таблиц (особенно таблицу T), так что если не чистить схему при переходе к очередной главе, может возникнуть конфликт.

Кроме того, если попытаться вручную удалять объекты, созданные в примере (а не просто удалять схему оператором **drop user ИМЯ_ПОЛЬЗОВАТЕЛЯ cascade),** нужно учитывать, что в именах Java-объектов используются символы разных регистров. Так что, если выполнить пример из главы 19:

```
tkyte@TKYTE816> create or replace and compile
  2    Java source named "demo"
  3    as
  4    import java.sql.SQLException;
```

то окажется, что для удаления необходимо выполнять оператор вида:

```
tkyte@TKYTE816> drop Java source "demo";

Java dropped.
```

Не забывайте использовать двойные кавычки вокруг идентификаторов Java-объектов, поскольку они создаются и хранятся с учетом регистра.

1

Разработка успешных приложений для Oracle

Значительную часть времени я провожу, работая с программным обеспечением СУБД Oracle или, точнее, с людьми, которые это программное обеспечение используют. В течение последних двенадцати лет я работал над многими проектами, как успешными, так и закончившимися неудачно, и если бы потребовалось обобщить приобретенный при этом опыт несколькими фразами, я бы сказал следующее:

- успех или неудача разработки приложения базы данных (приложения, зависящего от базы данных) определяется тем, как оно использует базу данных;
- в команде разработчиков должно быть ядро "программистов базы данных", обеспечивающих согласованность логики работы с базой данных и настройку производительности системы.

Эти утверждения могут показаться очевидными, но опыт показывает, что слишком многие используют СУБД как "черный ящик", о деталях устройства которого знать необязательно. Они могут использовать генератор SQL, позволяющий не затруднять себя изучением языка SQL. Возможно, они решат использовать базу данных как обычный файл с возможностью чтения записей по ключу. Как бы то ни было, я могу вам сказать, что подобного рода соображения почти наверняка приводят к неправильным выводам — работать, не понимая устройства СУБД, просто нельзя. В этой главе описано, **почему** необходимо знать устройство СУБД, в частности, почему необходимо понимать:

- архитектуру СУБД, ее компоненты и алгоритмы работы;
- что такое средства управления одновременным доступом и каково их значение для разработчиков;

- как настраивать приложение с первого дня его создания;
- как реализованы определенные компоненты СУБД и чем эта реализация отличается от обычно предполагаемой;
- какие возможности реализованы в самой СУБД и почему, как правило, лучше использовать предоставляемые СУБД функции, а не реализовать их самостоятельно;
- зачем может понадобиться более глубокое знание языка SQL.

Этот список тем для начального изучения может показаться слишком длинным, но давайте рассмотрим следующую аналогию: если бы вы разрабатывали масштабируемое, высокопроизводительное приложение для абсолютно новой операционной системы (ОС), с чего бы вы начали? Надеюсь, ваш ответ: "С изучения особенностей функционирования этой новой ОС, работы приложений в ней и т.п.". Если ответ принципиально другой, ваша разработка обречена на неудачу.

Рассмотрим, например, одну из ранних версий Windows (скажем, Windows 3.x). Она, как и ОС UNIX, была "многозадачной" операционной системой. Однако эта многозадачность была не такой, как в ОС UNIX, — использовалась модель невытесняющей многозадачности (т.е., если работающее приложение не возвращает управление, ничто другое работать не может, включая операционную систему). Фактически, по сравнению с UNIX, Windows 3.x вообще не была многозадачной ОС. Для создания эффективных приложений разработчики должны были точно знать, как реализована возможность "многозадачности" Windows. Если необходимо разрабатывать приложение, работающее непосредственно в среде определенной ОС, понимание особенностей этой ОС очень важно.

То, что верно в отношении приложений, непосредственно работающих в среде операционной системы, верно и для приложений, работающих в среде СУБД: понимание особенностей СУБД является определяющим фактором успеха. Если вы не понимаете, что делает используемая СУБД или как она это делает, создаваемое приложение не будет работать успешно. Предположение о том, что успешно работающее в среде SQL Server приложение так же успешно будет работать и в среде Oracle, скорей всего не оправдается.

Мой подход

Прежде чем начать, хотелось бы объяснить вам мой подход к разработке. Я предпочитаю решать большинство проблем на уровне СУБД. Если что-то можно сделать в СУБД, я так и сделаю. Для этого есть две причины. Первая и главная состоит в том, что если встроить функциональность в СУБД, то ее можно будет **применять где угодно. Я** не знаю серверной операционной системы, для которой нет реализации СУБД Oracle. Одна и та же СУБД Oracle со всеми опциями работает везде — от Windows до десятков версий ОС UNIX и больших ЭВМ типа OS/390. Я часто разрабатываю и тестирую программы на моем портативном компьютере, где работает СУБД Oracle8/для Windows NT. А применяются эти программы на различных серверах с ОС UNIX, на которых работает та же версия СУБД. Если приходится реализовать функциональность за пределами

СУБД, ее очень сложно переносить на любую другую платформу. Одна из основных особенностей, делающих язык Java привлекательным для многих разработчиков, состоит в том, что программы всегда компилируются в одной и той же виртуальной среде, виртуальной машине Java Virtual Machine (JVM), и поэтому максимально переносимы. Именно эта особенность привлекает меня в СУБД. СУБД Oracle — это **моя** виртуальная машина, **моя** "виртуальная операционная система".

Мой подход состоит в том, чтобы делать в СУБД все, что возможно. Если требования выходят за пределы возможностей СУБД, я реализую соответствующие функции на языке Java вне СУБД. В этом случае особенности практически любой операционной системы скрываются. Мне все равно надо понимать, как работают **мои** "виртуальные машины" (Oracle или JVM) — надо знать используемые инструментальные средства, — но наиболее эффективная реализация соответствующих функций в конкретной ОС остается прерогативой создателей этих виртуальных машин.

Таким образом, зная лишь особенности работы одной "виртуальной ОС", можно создавать приложения, демонстрирующие отличную производительность и масштабируемость во многих операционных системах. Я не утверждаю, что можно полностью игнорировать базовую ОС, — просто разработчик приложений баз данных достаточно хорошо от нее изолирован, и ему не придется учитывать многие ее нюансы. Ваш АБД, отвечающий за поддержку СУБД Oracle, должен знать намного больше об особенностях базовой ОС (если не знает — найдите нового АБД!). При разработке клиент-серверного программного обеспечения, если основная часть кода вынесена из СУБД и виртуальной машины (наиболее популярной виртуальной машиной, вероятно, является Java Virtual Machine), разработчику придется учитывать особенности ОС сервера.

При разработке приложений баз данных я использую очень простую мантру:

- если можно, сделай это с помощью одного оператора SQL;
- если это нельзя сделать с помощью одного оператора SQL, сделай это в PL/SQL;
- если это нельзя сделать в PL/SQL, попытайся использовать хранимую процедуру на языке Java;
- если это нельзя сделать в Java, сделай это в виде внешней процедуры на языке C;
- если это нельзя реализовать в виде внешней процедуры на языке C, надо серьезно подумать, зачем это вообще делать...

В книге вы увидите применение этого подхода. Мы будем использовать язык PL/SQL и его объектные типы для реализации того, что нельзя сделать в SQL. Язык PL/SQL существует давно, за ним стоит более тринадцати лет настройки, и нет другого языка, настолько тесно интегрированного с языком SQL и настолько оптимизированного для взаимодействия с SQL. Когда возможностей PL/SQL оказывается недостаточно, например, при доступе к сети, отправке сообщений электронной почты и т.п., мы будем использовать язык Java. Иногда мы будем решать определенные задачи с помощью языка C, но обычно лишь в тех случаях, когда программирование на C — единственно возможный вариант или когда обеспечиваемая компилятором C скорость работы программы действительно необходима. Во многих случаях сейчас последняя причина отпадает при использовании компиляции в машинные коды программ на языке Java (возможности преобразовать байт-код Java в специфический объектный код операционной системы

для данной платформы). Это обеспечивает программам на Java такую же скорость работы, как и у программ на языке C.

Подход с использованием принципа черного ящика

У меня есть предположение, основанное на личном опыте, почему так часто разработка приложений баз данных заканчивается неудачно. Позвольте уточнить, что к разряду неудавшихся разработок я отношу также проекты, официально не признанные неудавшимися, но потребовавшие на разработку и внедрение намного больше времени, чем планировалось первоначально, поскольку пришлось их существенно "переписывать", "перепроектировать" или "настраивать". Лично я такие не завершенные в строк проекты считаю неудавшимися: очень часто их вполне можно было завершить вовремя (и даже досрочно).

Наиболее типичной причиной неудачи является нехватка практических знаний по используемой СУБД — элементарное непонимание основ работы используемого инструментального средства. Подход по принципу "черного ящика" требует осознанного решения: оградить разработчиков от СУБД. Их заставляют не вникать ни в какие особенности ее функционирования. Причины использования этого подхода связаны с опасениями, незнанием и неуверенностью. Разработчики слышали, что СУБД — это "сложно", язык SQL, транзакции и целостность данных — не менее "сложно". Решение: не заставлять никого делать что-либо "сложное". Будем относиться к СУБД, как к черному ящику, и найдем инструментальное средство, которое сгенерирует необходимый код. Изолируем себя несколькими промежуточными уровнями, чтобы не пришлось сталкиваться непосредственно с этой "сложной" СУБД.

Такой подход к разработке приложений баз данных я не мог понять никогда. Одна из причин, почему мне трудно это понять, состоит в том, что для меня изучение языков Java и C оказалось намного сложнее, чем изучение основ работы СУБД. Я сейчас очень хорошо знаю языки Java и C, но для их освоения мне понадобилось намного больше практического опыта, чем для достижения соответствующего уровня компетентности при использовании СУБД. В случае СУБД необходимо знать, как она работает, но детали знать необязательно. При программировании на языке C или Java, необходимо, например, знать все особенности используемых компонентов; кроме тою, это очень **большие по объему** языки.

Еще одна причина — то, что при создании приложений базы данных **самым важным компонентом программного обеспечения является СУБД.** Для успешной разработки необходимо учитывать это и доводить до сведения разработчиков, постоянно обращая на это их внимание. Много раз я сталкивался с проектами, где придерживались прямо противоположных воззрений.

Вот типичный сценарий такого рода разработки.

- Разработчики были полностью обучены графической среде разработки или соответствующему языку программирования (например, Java), использованных для создания клиентской части приложения. Во многих случаях они обучались несколько недель, если не месяцев.

- Команда разработчиков ни одного часа не изучала СУБД Oracle и не имела никакого опыта работы с ней. Многие разработчики вообще впервые сталкивались с СУБД.
- В результате разработчики столкнулись с огромными проблемами, связанными с производительностью, обеспечением целостности данных, зависанием приложений и т.д. (но пользовательский интерфейс выглядел отлично).

Не сумев обеспечить нужную производительность, разработчики обращались за помощью ко мне. Особенно показателен один случай. Я не мог вспомнить точный синтаксис новой команды, которую надо было использовать, и попросил руководство *SQL Reference*. Мне принесли экземпляр из документации по СУБД Oracle версии 6.0, хотя разработка велась на версии 7.3, через пять лет после выхода версии 6.0! Ничего другого для работы у них не было, но это вообще никого не беспокоило. Хотя необходимое им для трассировки и настройки инструментальное средство в то время вообще не существовало. Хотя за пять лет, прошедших после написания имевшейся у них документации, были добавлены такие средства, как триггеры, хранимые процедуры, и многие сотни других. Несложно понять, почему им потребовалась помощь, гораздо труднее было решить их проблемы.

Странная идея о том, что разработчик **приложения баз данных** должен быть ограждены от СУБД, чрезвычайно живуча. Многие почему-то считают, что разработчикам не следует тратить время на изучение СУБД. Неоднократно приходилось слышать: "СУБД Oracle — самая масштабируемая в мире, моим сотрудникам не нужно ее изучать, потому что СУБД со всеми проблемами справится сама". Действительно, СУБД Oracle — самая масштабируемая. Однако написать плохой код, который масштабироваться не будет, в Oracle намного проще, чем написать хороший, масштабируемый код. Можно заменить СУБД Oracle любой другой СУБД — это утверждение останется верным. Это факт: проще писать приложения с низкой производительностью, чем высокопроизводительные приложения. Иногда очень легко создать однопользовательскую систему на базе самой масштабируемой СУБД в мире, если не знать, что делаешь. СУБД — это инструмент, а неправильное применение любого инструмента может привести к катастрофе. Вы будете щипцами колоть орехи так же, как молотком? Можно, конечно, и так, но это неправильное использование инструмента, и результат вас не порадует. Аналогичные результаты будут и при игнорировании особенностей используемой СУБД.

Я недавно работал над проектом, в котором проектировщики придумали очень элегантную архитектуру. Клиент с помощью Web-браузера взаимодействовал по протоколу HTTP с сервером приложений, обеспечивающим поддержку Java Server Pages (JSP). Алгоритмы работы приложения целиком генерировались инструментальными средствами и реализовывались в виде компонентов EJB (с использованием постоянного хранения на базе контейнеров), причем физически они выполнялись другим сервером приложений. В базе данных хранились только таблицы и индексы.

Итак, мы начали с технически сложной архитектуры. Для решения задачи должны взаимодействовать друг с другом четыре компонента. Web-браузер получает страницы JSP от сервера приложений, который обращается к компонентам EJB, а те, в свою очередь, — к СУБД. Для разработки, тестирования, настройки и внедрения этого приложения необходимы были технически компетентные специалисты. После завершения раз-

работки меня попросили оценить производительность приложения. Прежде всего я хотел узнать подход разработчиков к СУБД:

- где, по их мнению, у приложения могут быть узкие места, точки потенциальных конфликтов?
- каковы, по их мнению, основные препятствия для достижения требуемой производительности?

Они не имели ни малейшего представления об этом. На вопрос о том, кто поможет мне переписать код компонента EJB для настройки сгенерированного запроса, ответ был следующий: "О, этот код нельзя изменять, все надо делать в базе данных". То есть, приложение должно оставаться неизменным. В этот момент я был готов отказаться от работы над проектом — ясно, что заставить это приложение нормально работать невозможно:

- приложение было создано без учета масштабирования на уровне базы данных;
- приложение нельзя настраивать и вообще изменять;
- по моему опыту, от 80 до 90 процентов **всей** настройки выполняется на уровне приложения, а не на уровне базы данных;
- разработчики не представляли себе, что делают их компоненты с базой данных и где искать потенциальные проблемы.

Все это стало понятно уже через час тестирования. Как оказалось, в приложении сначала выполнялся оператор:

```
select * from t for update;
```

Это приводило к строго последовательной работе **всех** клиентов. В базе данных была реализована такая модель, что перед выполнением любых существенных действий приходилось блокировать весьма большой ресурс. Это моментально превращало приложение в очень большую однопользовательскую систему. Разработчики не верили мне (в другой СУБД, использующей разделяемую блокировку чтения, наблюдалась другая ситуация). После десяти минут работы с инструментальным средством TKPROF (о нем подробно написано в главе 10) я смог продемонстрировать, что именно этот оператор SQL выполнялся приложением (они об этом не знали — просто никогда не видели генерируемых операторов SQL). Я не просто показал, какие операторы SQL выполняются приложением, но и с помощью пары сеансов SQL*Plus продемонстрировал, что второй сеанс не начинается до полного завершения работы первым сеансом.

Итак, вместо того, чтобы неделю тестировать производительность приложения, я употребил это время на обучение разработчиков настройке, особенностям блокирования в базах данных, механизмам управления одновременным доступом, сравнение их реализаций в СУБД Oracle, Informix, SQL Server, DB2 и так далее (во всех этих СУБД они различны). Но сначала мне пришлось понять, однако, почему использовался оператор **SELECT FOR UPDATE.** Оказалось, что разработчики хотели добиться повторяемости при чтении.

Повторяемость при чтении — это такой режим работы приложения, когда повторное чтение в транзакции строки, которая уже один раз прочитана в этой транзакции, дает тот же результат.

Зачем им это было нужно? Они слышали, что "это хорошо". Ладно, предположим, повторяемость при чтении действительно необходима. В СУБД Oracle это делается путем установки уровня изолированности транзакции SERIALIZABLE (что дает не только повторяемость при чтении строки, но и повторяемость при выполнении любого запроса — если два раза выполнить один и тот же запрос в пределах транзакции с таким уровнем изолированности, будут получены одинаковые результаты). Для обеспечения повторяемости при чтении в Oracle не нужно использовать SELECT FOR UPDATE — это делается только для обеспечения последовательного доступа к данным. К сожалению, использованное разработчиками инструментальное средство это не учитывало — оно было создано для использования с другой СУБД, где именно так повторяемость при чтении и достигалась.

Итак, в данном случае для установки уровня изолированности транзакций SERIALIZABLE пришлось создать триггер на регистрацию в базе данных, изменяющий параметры сеанса (уровень изолированности транзакций) для данного приложения. Затем мы отключили все установки повторяемости при чтении в использовавшемся инструментальном средстве и повторно запустили приложение. Теперь, без конструкции FOR UPDATE, в базе данных определенные действия стали выполняться одновременно.

Это была далеко не последняя проблема данного проекта. Нам пришлось разобраться:

- как настраивать операторы SQL, не изменяя их кода (это сложно — некоторые методы мы рассмотрим в главе 11);
- как измерять производительность;
- как находить узкие места;
- что и как индексировать, и так далее.

В конце недели разработчики, никогда ранее не работавшие с СУБД, были удивлены тем, что в действительности она дает возможность сделать, как легко получить указанную выше информацию и, что наиболее важно, как существенно все это может сказаться на производительности приложения. Тестированием производительности в течение этой недели мы не занимались (им кое-что пришлось переделывать!), но в конечном итоге проект завершился успешно — просто на пару недель позже запланированного срока.

Это не критика инструментальных средств или современных технологий, таких как компоненты EJB и поддержка постоянного существования на базе контейнеров. Это — критика намеренного игнорирования особенностей СУБД, принципов ее работы и использования. Технологии, выбранные в этом проекте, работали отлично, но лишь после того, как разработчики немного разобрались в самой СУБД.

Подводя итоги: СУБД — это краеугольный камень приложения. Если она не работает как следует, все остальное не имеет значения. Если плохо работает черный ящик, что

с ним делать? Его нельзя исправить, нельзя настроить (поскольку непонятно, как он устроен), и такую позицию вы выбрали сами. Но есть и другой подход, который я отстаиваю: разберитесь в используемой СУБД и принципах ее работы, поймите, что она может делать, и используйте весь ее потенциал.

Как надо (и как не надо) разрабатывать приложения баз данных

Однако хватит теоретизировать, по крайней мере пока. В оставшейся части главы я использую более эмпирический подход, описывая, почему знание СУБД и особенностей ее работы существенно повышает шансы успешной реализации приложения (без необходимости переписывать его дважды!). Некоторые проблемы решить легко, если известно, как их найти. Для решения других потребуются существенные переделки. Одна из целей этой книги — помочь вам вообще избежать проблем.

В следующих разделах я опишу ряд важнейших особенностей СУБД Oracle, не вдаваясь в подробности их реализации и использования. Например, я опишу только одно из последствий использования архитектуры многопотокового сервера (Multi- Threaded Server — MTS) — режима, в котором можно (а иногда и нужно) конфигурировать сервер Oracle для поддержки множества пользовательских сеансов. Я, однако, не буду детально описывать архитектуру MTS, особенности ее работы и т.п. Все это подробно описано в руководстве Oracle Server Concepts Manual (дополнительную информацию можно найти также в руководстве Net8 Administrators Guide).

Понимание архитектуры СУБД Oracle

Недавно я участвовал в проекте, создатели которого решили использовать только новейшие, самые совершенные технологии: все программное обеспечение было написано на Java в виде компонентов EJB. Клиентское приложение взаимодействовало с СУБД через эти компоненты — никакого протокола Net8. Между клиентом и сервером не передавались операторы SQL — только обращения к компонентам EJB с помощью вызовов удаленных методов (Remote Method Invocation — RMI) по протоколу Internet Inter-Orb Protocol (HOP).

Об организации RMI по протоколу ПОР можно узнать на сайте ***http://java.sun.com/products/rmi-nop/.***

Это вполне допустимый подход. Такой способ взаимодействия работает и может быть весьма масштабируемым. Те, кто разрабатывал архитектуру, хорошо понимали язык Java, технологию компонентов EJB, знали используемые протоколы, в общем — всю кухню. Им казалось, что имеются все основания для успешной реализации подобного проекта. Когда же выяснилось, что приложение может поддерживать лишь нескольких пользователей, они решили, что проблема — в СУБД, и усомнились в декларируемой корпорацией Oracle "рекордной масштабируемости СУБД".

Проблема, однако, была не в СУБД, а в незнании особенностей ее работы, и некоторые решения, принятые на этапе проектирования, привели к краху приложения в целом. Чтобы использовать компоненты EJB в базе данных, сервер Oracle должен быть сконфигурирован для работы в режиме многопотокового (MTS), а не выделенного сервера. Чего не понимала команда разработчиков в принципе, так это последствий использования режима MTS в сочетании с компонентами EJB для их приложения. При отсутствии этого понимания, как и знания основ работы СУБД Oracle вообще, были приняты два ключевых решения.

- В компонентах будут запускаться хранимые процедуры, работающие по 45 и более секунд (иногда — намного дольше).
- Связываемые переменные использоваться не будут. В условиях всех запросов будут использоваться литеральные константы. Все входные данные для процедур будут передаваться в виде строк. Это "проще", чем использовать связываемые переменные.

Эти, казалось бы, непринципиальные решения обусловили неизбежный провал проекта. Все было сделано так, что предельно масштабируемая СУБД не справлялась с нагрузкой даже при небольшом количестве пользователей. Нехватка знаний об особенностях работы СУБД свела на нет все глубокие знания разработчиков по созданию компонентов на Java и распределенной обработке. Если бы они нашли время на минимальное изучение особенностей работы СУБД Oracle и затем применили два представленных далее простых принципа, шансы на успех проекта уже в первой версии существенно бы повысились.

Избегайте длительных транзакций в среде MTS

Решение использовать транзакции продолжительностью более 45 секунд в среде MTS выдало недостаточное понимание назначения режима MTS и особенностей его работы в Oracle. Если коротко, в режиме MTS используется общий пул серверных процессов, обслуживающий намного больший пул конечных пользователей. Это похоже на пул подключений. Поскольку создание и управление процессом — наиболее дорогостоящие операции, выполняемые операционной системой, режим MTS дает большие преимущества для крупномасштабной системы. Можно обслуживать 100 пользователей всего пятью или десятью разделяемыми серверными процессами.

Когда разделяемый серверный процесс получает запрос на изменение данных или выполнение хранимой процедуры, он привязывается к этой задаче до ее завершения. Ни одна другая задача не будет использовать разделяемый серверный процесс, пока не будет закончено изменение или не завершится выполнение хранимой процедуры. Поэтому при использовании режима MTS надо применять как можно быстрее выполняющиеся операторы. Режим MTS создан для обеспечения масштабируемости систем оперативной обработки транзакций (OOT), для которых характерны операторы, выполняющиеся за доли секунды. Речь идет об изменениях отдельных строк, вставке нескольких строк и запросах записей по первичному ключу. Не стоит в этом режиме выполнять пакетные процессы, для завершения которых требуются десятки секунд или минуты.

Если все операторы выполняются быстро, архитектура MTS работает отлично. Можно эффективно обслуживать небольшим количеством процессов большое сообщество пользователей. Если же имеются сеансы, монополизирующие разделяемый сервер надолго, то кажется, что СУБД "зависает". Пусть сконфигурировано десять разделяемых серверов для поддержки 100 пользователей. Если в некоторый момент времени десять пользователей одновременно введут оператор, выполняющийся более 45 секунд, то всем остальным транзакциям (и новым подключениям) придется ждать. Если некоторым из ожидающих в очереди сеансов необходимо выполнять оператор такой же продолжительности, возникает большая проблема — "зависание" будет продолжаться не 45 секунд, а намного дольше. Даже если желающих выполнить подобный оператор одновременно будет не десять, а лишь несколько, все равно будет наблюдаться существенное падение производительности сервера. Мы отберем на длительное время совместно используемый ресурс, и это плохо. Вместо десяти серверных процессов, выполняющих быстрые запросы в очереди, остается пять или шесть (или еще меньше). Со временем система станет работать с производительностью, заметно меньше предполагаемой, исключительно из-за нехватки этого ресурса.

Простое решение "в лоб" состоит в запуске большего количества разделяемых серверов, но в конечном итоге придется запускать разделяемый сервер для каждого пользователя, а это неприемлемо для системы с тысячами пользователей (как та, что создавалась в рассматриваемом проекте). Это не только создает узкие места в самой системе (чем большим количеством процессов приходится управлять, тем больше процессорного времени на это уходит), но и просто не соответствует целям создания режима MTS.

Реальное решение этой проблемы оказалось простым: не выполнять продолжительные транзакции на сервере, работающем в режиме MTS. А вот реализация этого решения оказалась сложнее. Это можно было сделать несколькими способами, но все они требовали существенных изменений архитектуры. Самым подходящим способом, требующим минимальных изменений, оказалось использование средств расширенной поддержки очередей (Advanced Queues — AQ).

AQ — это промежуточное программное обеспечение для обмена сообщениями, реализованное на базе СУБД Oracle и позволяющее клиентскому сеансу добавлять сообщения в таблицу очереди базы данных. Это сообщение в дальнейшем (обычно сразу после фиксации транзакции) выбирается из очереди другим сеансом, проверяющим содержимое сообщения. Сообщение содержит информацию для обработки другим сеансом. Оно может использоваться для эмуляции мгновенного выполнения за счет вынесения продолжительного процесса за пределы интерактивного клиента.

Итак, вместо выполнения 45-секундного процесса, компонент должен помешать запрос со всеми необходимыми входными данными в очередь и выполнять его асинхронно, а не синхронно. В этом случае пользователю не придется ждать ответа 45 секунд, то есть система становится более динамичной.

Хотя, судя по описанию, этот подход прост (подключение механизма AQ полностью решает проблему), потребовалось сделать намного больше. Этот 45-секундный процесс генерировал идентификатор транзакции, необходимый на следующем шаге в интерфейсе для соединения таблиц — по проекту интерфейс без этого не работал. Используя меха-

низм AQ, мы не ждем генерации идентификатора транзакции, — мы обращаемся к системе с просьбой сделать это когда-нибудь. Поэтому приложение опять оказалось в тупике. С одной стороны, мы не можем ждать завершения процесса 45 секунд, но, с другой стороны, для перехода к следующему экрану необходим сгенерированный идентификатор, а получить его можно только спустя 45 секунд. Для того чтобы решить эту проблему, пришлось синтезировать собственный поддельный идентификатор транзакции, изменить продолжительный процесс так, чтобы он принимал этот сгенерированный поддельный идентификатор и обновлял таблицу, записывая его по завершении работы, благодаря чему реальный идентификатор транзакции связывался с поддельным. То есть, вместо получения реального идентификатора в результате длительного процесса, этот идентификатор становится для процесса входными данными. Во всех "подчиненных" таблицах использовался этот поддельный идентификатор транзакции, а не реальный (поскольку генерации реального надо ждать определенное время). Нам также пришлось пересмотреть использование этого идентификатора транзакции, чтобы понять, как это изменение повлияет на другие модули, и так далее.

Еще одна проблема состояла в том, что при синхронной работе, если 45-секундный процесс завершался неудачно, пользователь узнавал об этом сразу. Он мог устранить причину ошибки (обычно путем изменения входных данных) и повторно выполнить запрос. Теперь, когда транзакции выполняются асинхронно с помощью механизма AQ, сделать это невозможно. Для поддержки отсроченного уведомления об ошибке пришлось добавить новые средства. В частности, понадобилось реализовать механизм потоков заданий для отправки информации о неудавшихся транзакциях соответствующему лицу.

В результате пришлось существенно пересмотреть структуру базы данных. Пришлось добавить новое программное обеспечение (AQ). Пришлось также создать новые процессы (управление потоками заданий и другие служебные процессы). К положительным последствиям этих изменений можно отнести не только решение проблемы с архитектурой MTS, но и удобство для пользователя (создавалась видимость более быстрой реакции системы). С другой стороны, все эти изменения существенно задержали завершение проекта, поскольку проблемы были выявлены лишь непосредственно перед внедрением, на этапе тестирования масштабируемости. Очень жаль, что приложение сразу не был правильно спроектировано. Если бы разработчики знали, как физически реализован механизм MTS, было бы ясно, что исходный проект не обеспечивает требуемой масштабируемости.

Используйте связываемые переменные

Если бы мне пришлось писать книгу о том, как создавать немасштабируемые приложения Oracle, первая и единственная ее глава называлась бы *"Не используйте связываемые переменные"*. Это — основная причина проблем, связанных с производительностью, и основная помеха масштабируемости. Особенности использования разделяемого пула Oracle (очень важной структуры данных разделяемой памяти) требуют от разработчиков использовать связываемые переменные. Если надо замедлить работу приложения Oracle, вплоть до полного останова, — откажитесь от их использования.

Связываемая переменная — это подставляемый параметр запроса. Например, для получения записи доя сотрудника с номером 123, можно выполнить запрос:

```
select * from emp where empno = 123;
```

Но можно задать и другой запрос:

```
select * from emp where empno = :empno;
```

В обычной системе информацию о сотруднике с номером 123 могут запрашивать всего один раз. В дальнейшем будут запрашивать информацию о сотрудниках с номерами 456, 789 и т.д. При использовании в запросе литералов (констант) каждый запрос является для СУБД абсолютно новым, никогда ранее не выполнявшимся. Его надо разбирать, уточнять (определять объекты, соответствующие именам), проверять права доступа, оптимизировать и т.д. — короче, каждый выполняемый уникальный оператор придется компилировать при каждом выполнении.

Во втором запросе используется связываемая переменная, :empno, значение которой подставляется в запрос при выполнении. Этот запрос компилируется один раз, а затем план его выполнения запоминается в разделяемом пуле (в библиотечном кэше), из которого его можно выбрать для повторного выполнения. Различие между этими двумя вариантами в плане производительности и масштабируемости — огромное, даже принципиальное.

Из представленного выше описания вполне понятно, что разбор оператора с явными, жестко заданными константами (так называемый *жесткий разбор)* выполняется дольше и требует намного больше ресурсов, чем повторное использование уже сгенерированного плана запроса (его называют *мягким разбором).* Менее очевидным может оказаться, насколько постоянный жесткий разбор сокращает количество пользователей, поддерживаемых системой. Отчасти это связано с повышенным потреблением ресурсов, но в гораздо большей степени — с механизмом защелок, используемых в библиотечном кэше. При жестком разборе запроса СУБД будет дольше удерживать определенные низкоуровневые средства обеспечения последовательного доступа, которые называются *защелками* (подробнее о них см. в главе 3). Защелки защищают структуры данных в разделяемой памяти сервера Oracle от одновременного изменения двумя сеансами (иначе эти структуры данных Oracle в конечном итоге были бы повреждены) и от чтения этой структуры данных по ходу изменения другим сеансом. Чем чаще и на более продолжительное время на эти структуры данных устанавливаются защелки, тем длиннее становится очередь для установки этих защелок. Точно так же происходит при использовании длинных транзакций в среде MTS, — монополизируются критические ресурсы. Временами машина может казаться минимально загруженной, а СУБД работает очень медленно. Вполне вероятно, что один из сеансов удерживает защелку и формируется очередь в ожидании ее освобождения. В результате работа с максимальной скоростью невозможна. Достаточно одного неверно работающего приложения для существенного снижения производительности всех остальных приложений. Одно небольшое приложение, не использующее связываемые переменные, приводит со временем к удалению из разделяемого пула необходимых SQL-операторов других хорошо настроенных приложений. Достаточно ложки дегтя, чтобы испортить бочку меда.

При использовании связываемых переменных любой сеанс, выдающий тот же самый запрос, будет использовать уже скомпилированный план выполнения из библиотечного кэша. Подпрограмма компилируется один раз, а используется многократно. Это очень эффективно, и именно такую работу пользователей предполагает СУБД. При этом не

только используется меньше ресурсов (мягкий разбор требует намного меньше ресурсов), но и защелки удерживаются значительно меньше времени, и нужны гораздо реже. Это повышает производительность и масштабируемость.

Чтобы хоть примерно понять, насколько существенно это может сказаться на производительности, достаточно выполнить очень простой тест:

```
tkyte@TKYTE816> alter system flush shared_pool;

System altered.
```

Здесь я начинаю с "пустого" разделяемого пула. Если потребуется выполнять этот тест многократно, придется очищать разделяемый пул каждый раз, иначе представленный ниже оператор SQL, в котором не используются связываемые переменные, окажется в кэше и будет выполняться очень быстро.

```
tkyte@TKYTE816> set timing on
tkyte@TKYTE816> declare
  2        type rc is ref cursor;
  3    l_rc    rc;
  4        l_dummy all_objects.object_name%type;
  5        l_start number default dbms_utility.get_time;
  6  begin
  7      for i in 1 .. 1000
  8       loop
  9            open l_rc for
 10            'select object_name
 11               from all_objects
 12              where object_id = ' || i;
 13            fetch l_rc into l_dummy;
 14            close l_rc;
 15       end loop;
 16       dbms_output.put_line
 17       (round((dbms_utility.get_time-l_start)/100, 2)  ||
 18         ' seconds...');
 19  end;
 20  /
14.86 seconds...

PL/SQL procedure successfully completed.
```

В этом коде используется динамический SQL для запроса одной строки из таблицы **ALL_OBJECTS**. Он генерирует 1000 уникальных запросов со значениями 1, 2, 3, ... и так далее, жестко заданными в конструкции WHERE. На моем ноутбуке с процессором Pentium 300 Мгц для его выполнения потребовалось около 15 секунд (скорость выполнения на разных машинах может быть различной).

Теперь сделаем то же самое с использованием связываемых переменных:

```
tkyte@TKYTE816> declare
  2        type rc is ref cursor;
  3    l_rc    rc;
  4        l_dummy all_objects.object_name%type;
  5        l_start number default dbms_utility.get_time;
  6  begin
  7      for i in 1 .. 1000
```

```
 8        loop
 9             open l_rc for
10             'select object_name
11                from all_objects
12               «here object_id = :x'
13             using i;
14             fetch l_rc into l_dummy;
15             close l_re;
16        end loop;
17        dbms_output.put_line
18        (round((dbms_utility.get_time-l_start)/100, 2)  ||
19      '      seconds...');
20 end;
21  /
1.27 seconds...
```

PL/SQL procedure successfully completed.

В этом коде использован точно такой же алгоритм. Единственное изменение — вместо жестко заданных значений 1, 2, 3... и так далее в запросе используется связываемая переменная. Результаты весьма впечатляющи. Код не только выполняется намного быстрее **(разбор** запросов требовал больше времени, чем их реальное **выполнение!),** но и позволяет большему количеству пользователей одновременно работать с системой.

Выполнение операторов SQL без связываемых переменных во многом подобно перекомпиляции подпрограммы перед каждым вызовом. Представьте себе передачу клиентам такого исходного кода на языке Java, что перед любым вызовом метода класса им необходимо вызывать компилятор Java, компилировать класс и выбрасывать сгенерированный байт-код сразу после выполнения метода. При необходимости вызова того же метода в дальнейшем им пришлось бы делать то же самое — компилировать, запускать и выбрасывать байт-код. В своих приложениях никто так не поступает — не делайте этого и в СУБД.

В главе 10 мы рассмотрим способы определить, используются ли связываемые переменные, различные варианты их применения, поддерживаемую СУБД возможность автоматической подстановки связываемых переменных и т.д. Мы также рассмотрим особый случай, когда использование связываемых переменных нежелательно.

Часто, как и в рассматриваемом проекте, — переписывание существующего кода так, чтобы использовались связываемые переменные, является единственно возможным выходом. Получаемый в результате код работает на несколько порядков быстрее и во много раз увеличивается количество поддерживаемых системой одновременно работающих пользователей. Для этого, однако, требуется много времени и усилий. Дело не в том, что использовать связываемые переменные сложно или при этом часто делают ошибки, — проблема в том, что с самого начала этого не делали, и поэтому пришлось пересмотреть и изменить практически **весь** код. Разработчикам не пришлось бы платить такую цену, если бы они с первого дня понимали принципиальную важность использования в приложении связываемых переменных.

Особенности управления одновременным доступом

Управление одновременным доступом — это то, чем отличаются различные СУБД. Именно это отличает СУБД от файловой системы и одну СУБД от другой. Для программиста важно, чтобы его приложение базы данных корректно работало в условиях одновременного доступа, и именно это постоянно забывают проверять. Приемы, прекрасно работающие в условиях последовательного доступа, работают гораздо хуже при одновременном их применении несколькими сеансами. Если не знать досконально, как в конкретной СУБД реализованы механизмы управления одновременным доступом, то:

- будет нарушена целостность данных;
- приложение будет работать медленнее, чем предусмотрено, даже при небольшом количестве пользователей;
- будет потеряна возможность масштабирования до большого числа пользователей.

Обратите внимание: я не пишу "возможно, будет..." или "вы рискуете..." — все эти проблемы точно будут. Все это вы получите, даже не представляя, что именно происходит. При неправильном управлении одновременным доступом будет нарушена целостность данных, поскольку то, что работает отдельно, будет работать не так, как предполагалось, в многопользовательской среде. Приложение будет работать медленнее, поскольку придется ждать доступа к данным. Возможность масштабирования будет потеряна из-за проблем с блокированием и конфликтов блокировок. По мере усложнения запросов к ресурсу ждать придется все дольше и дольше. Можно провести аналогию с таможенным переходом на границе. Если машины приезжают по одной, равномерно, с предсказуемой частотой, никаких очередей нет. Если же одновременно приедет много машин, начинают формироваться очереди. Причем время ожидания растет нелинейно по отношению к длине очереди. С определенного момента больше времени сотрудников уходит на "наведение порядка" в очереди, чем на таможенный досмотр машин (в случае СУБД мы говорим о планировании процессов и переключении контекста).

Проблемы одновременного доступа выявлять сложнее всего — трудности сопоставимы с отладкой многопотоковой программы. Программа может отлично работать в управляемой, искусственной среде отладчика, но постоянно "слетать" в "реальном мире". Например, в условиях интенсивных обращений может оказаться, что два потока одновременно изменяют одну и ту же структуру данных. Такого рода ошибки очень сложно выявлять и исправлять. Если приложение тестировалось только в однопользовательском режиме, а затем внедряется в среде с десятками одновременно обращающихся пользователей, вполне вероятно проявление болезненных проблем с одновременным доступом.

В следующих двух разделах будет представлено два небольших примера того, как непонимание .особенностей управления одновременным доступом может разрушить данные или снизить производительность и масштабируемость приложения.

Реализация блокирования

СУБД использует блокировки, чтобы в каждый момент времени те или иные данные могли изменяться только одной транзакцией. Говоря проще, блокировки — это ме-

ханизм обеспечения одновременного доступа. **При** отсутствии определенной модели блокирования, предотвращающей одновременное изменение, например, одной строки, многопользовательский доступ к базе данных попросту невозможен. Однако при избыточном или неправильном блокировании одновременный доступ тоже может оказаться невозможным. Если пользователь или сама СУБД блокирует данные без необходимости, то работать одновременно сможет меньшее количество пользователей. Поэтому понимание назначения блокирования и способов его реализации в используемой СУБД принципиально важно для создания корректных и масштабируемых приложений.

Принципиально важно также понимать, что в различных СУБД блокирование реализовано по-своему. В одних — используется блокирование на уровне страниц, в других — на уровне строк; в некоторых реализациях выполняется эскалация блокировок со строчного на страничный уровень, в других — не выполняется; в некоторых СУБД используются блокировки чтения, в других — нет; в одних СУБД уровень изолированности транзакций **SERIALIZABLE** реализуется с помощью блокирования, а в других — через согласованные по чтению представления данных (без установки блокировок). Эти небольшие отличия могут перерасти в огромные проблемы, связанные с производительностью, или даже привести к возникновению ошибок в приложениях, если не понимать их особенностей.

Ниже приведены принципы блокирования в СУБД Oracle.

- Oracle блокирует данные на уровне строк и только при изменении. Эскалация блокировок до уровня блока или таблицы никогда не выполняется.
- Oracle никогда не блокирует данные с целью считывания. При обычном чтении блокировки на строки не устанавливаются.
- Сеанс, записывающий данные, не блокирует сеансы, читающие данные. Повторю: операции чтения **не блокируются** операциями записи. Это принципиально отличается от практически всех остальных СУБД, в которых операции чтения блокируются операциями записи.
- Сеанс записи данных блокируется, только если другой сеанс записи уже заблокировал строку, которую предполагается изменять. Сеанс считывания данных никогда не блокирует сеанс записи.

Эти факты необходимо учитывать при разработке приложений, однако следует помнить, что эти принципы используются только в Oracle. Разработчик, не понимающий, как используемая СУБД обеспечивает одновременный доступ, неизбежно столкнется с проблемами целостности данных (особенно часто это происходит, когда разработчик переходит с другой СУБД на Oracle, или наоборот, и не учитывает в приложении различия механизмов обеспечения одновременного доступа).

Один из побочных эффектов принятого в СУБД Oracle "неблокирующего" подхода состоит в том, что если действительно необходимо обеспечить доступ к строке не более чем одного пользователя в каждый момент времени, то именно разработчику необходимо предпринять для этого определенные усилия. Рассмотрим следующий пример. Один разработчик показывал мне только что завершенную им программу планирования ресурсов (учебных классов, проекторов и т.д.), находящуюся в стадии внедрения. Это приложение реализовало бизнес-правило, предотвращающее выделение ресурса более

чем одному лицу на любой период времени. То есть, приложение содержало специальный код, который проверял, что никто из пользователей не затребовал ресурс на тот же период времени (по крайней мере разработчик думал, что его код это проверяет). Код обращался к таблице планов и, если в ней не было строк с перекрывающимся временным интервалом, вставлял в нее новую строку. Итак, разработчик просто работал с парой таблиц:

```
create table resources(resource_name varchar2(25) primary key, . . . ) ;
create table schedules(resource_name varchar2(25) references resources,
                                    start_time date,
                                    end_time   date);
```

И прежде чем зарезервировать на определенный период, скажем, учебный класс, приложение выполняло запрос вида:

```
select count(*)
  from schedules
 where resource_name = :room_name
   and (start_tiroe between :new_start_time and :new_end_time
        or
        end_time between :new_start_time and :new_end_time)
```

Он казался разработчику простым и надежным: если возвращено значение 0, учебный класс можно занимать; если возвращено ненулевое значение, значит, учебный класс на этот период уже кем-то занят. Ознакомившись с используемым алгоритмом, я подготовил простой тест, показывающий, какая ошибка будет возникать при реальной эксплуатации приложения. Эту ошибку будет крайне сложно обнаружить и тем более установить ее причину, — кому-то может даже показаться, что это **ошибка СУБД.**

Я предложил его коллеге сесть за соседний компьютер, перейти на тот же экран и попросил на счет три обоих нажать на кнопку Go и попытаться зарезервировать класс на одно то же время. Оба смогли это сделать — то, что прекрасно работало в изолированной среде, не сработало в среде многопользовательской. Проблема в этом случае была вызвана неблокирующим чтением в Oracle. Ни один из сеансов не блокировал другой. Оба сеанса просто выполняли представленный выше запрос и применяли алгоритм резервирования ресурса. Они оба могли выполнять запрос, проверяющий занятость ресурса, даже если другой сеанс уже начал изменять таблицу планов (это изменение невидимо для других сеансов до фиксации, то есть до тех пор, когда уже слишком поздно). Поскольку сеансы никогда не пытались изменить одну и ту же строку в таблице планов, они никогда и не блокировали друг друга, вследствие чего бизнес-правило не срабатывало так, как ожидалось.

Разработчику необходим метод реализации данного бизнес-правила в многопользовательской среде, способ, гарантирующий что в каждый момент времени только один сеанс резервирует данный ресурс. В данном случае решение состояло в программном упорядочении доступа — кроме представленного выше запроса **count(*),** необходимо было сначала выполнить:

```
select * from resources where resource_name = :room_name FOR UPDATE;
```

Ранее в этой главе рассматривался пример, когда использование конструкции **FOR** UPDATE приводило к проблемам, но в этом случае она обеспечивает корректную ра-

боту бизнес-правила. Мы просто блокируем ресурс (учебный класс), использование которого планируется непосредственно **перед** его резервированием, до выполнения запроса к таблице планов, выбирающего строки для данного ресурса. Блокируя ресурс, который мы пытаемся зарезервировать, мы гарантируем, что никакой другой сеанс в это же время не изменяет план использования ресурса. Ему придется ждать, пока наша транзакция не будет зафиксирована — после этого он сможет увидеть сделанное в ней резервирование. Возможность перекрытия планов, таким образом, устранена. Разработчик должен понимать, что в многопользовательской среде иногда необходимо использовать те же приемы, что и при многопотоковом программировании. В данном случае конструкция **FOR UPDATE** работает как семафор. Она обеспечивает последовательный доступ к конкретной строке в таблице ресурсов, гарантируя, что два сеанса одновременно не резервируют ресурс.

Этот подход обеспечивает высокую степень параллелизма, поскольку резервируемых ресурсов могут быть тысячи, а мы всего лишь гарантируем, что сеансы изменяют конкретный ресурс поочередно. Это один из немногих случаев, когда необходимо блокирование вручную данных, которые не должны изменяться. Требуется уметь распознавать ситуации, когда это необходимо, и, что не менее важно, когда этого делать не нужно (пример, когда не нужно, приведен далее). Кроме того, такой прием не блокирует чтение ресурса другими сеансами, как это могло бы произойти в других СУБД, благодаря чему обеспечивается высокая масштабируемость.

Подобные проблемы приводят к масштабным последствиям при переносе приложения с одной СУБД на другую (к этой теме я вернусь чуть позже), которых разработчикам сложно избежать. Например, при наличии опыта разработки для другой СУБД, в которой пишущие сеансы блокируют читающих, и наоборот, разработчик может полагаться на это блокирование как защищающее от подобного рода проблем — именно так все и работает во многих СУБД, отличных от Oracle. В Oracle приоритет отдан одновременности доступа, и необходимо учитывать, что в результате все может работать по-другому.

В 99 процентах случаев блокирование выполняется незаметно, и о нем можно не заботиться. Но оставшийся 1 процент надо научиться распознавать. Для решения этой проблемы нет простого списка критериев типа "в таком-то случае надо сделать то-то". Нужно понимать, как приложение будет работать в многопользовательской среде и что оно будет делать в базе данных.

Многовариантность

Эта тема очень тесно связана с управлением одновременным доступом, поскольку создает основу для механизмов управления одновременным доступом в СУБД Oracle — Oracle использует модель многовариантной согласованности по чтению при одновременном доступе. В главе 3 мы более детально рассмотрим технические аспекты многовариантности, но по сути это механизм, с помощью которого СУБД Oracle обеспечивает:

- **согласованность по чтению для запросов:** запросы выдают согласованные результаты на момент начала их выполнения;
- **неблокируемые запросы:** запросы не блокируются сеансами, в которых изменяются данные, как это бывает в других СУБД.

Это две очень важные концепции СУБД Oracle. Термин *многовариантность* произошел от того, что фактически СУБД Oracle может одновременно поддерживать множество версий данных в базе данных. Понимая сущность многовариантности, всегда можно понять результаты, получаемые из базы данных. Наиболее простой из известных мне способов **продемонстрировать** многовариантность в Oracle:

```
txyte@TKYTE816> create table t
  2  as
  3  select * from all_users;
Table created.

tkyte@TKYTE816> variable x refcursor
tkyte@TKYTE816> begin
  2           open :x for select * from t;
  3  end;

  4  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> delete from t;

18 rows deleted.

txyte@TKYTE816> commit;

Commit complete.

tkyte@TKYTE816> print x

USERNAME                       USER_ID  CREATED
SYS                                  0  04-NOV-00
SYSTEM                               5  04-KOV-00
DBSNMP                              16  04-HOV-00
AURORA$ORB$UNAUTHENTICATED          24  04-NOV-00
ORDSYS                              25  04-NOV-00
ORDPLUGIMS                          26  04-NOV-00
MDSYS                               27  04-NOV-00
CTXSYS                              30  04-NOV-OO

DEMO                                57  07-PEB-01
18 rows selected.
```

В этом примере мы создали тестовую таблицу T и заполнили ее данными из представления ALL_USERS. Мы открыли курсор для этой таблицы. Мы **не выбирали данные** с помощью этого курсора, просто открыли его.

Помните, что при открытии курсора сервер Oracle не "отвечает" на запрос; он никуда не копирует данные при открытии курсора (представьте, сколько времени потребовало бы открытие курсора для таблицы с миллиардом строк в противном случае). Курсор просто открывается и дает результаты запроса по ходу обращения к данным. Другими словами, он будет читать данные из таблицы при извлечении их через курсор.

В том же (или в другом) сеансе мы затем удаляем все данные из таблицы. Более того, мы даже фиксируем **(COMMIT)** это удаление. Строк больше нет — не так ли? На самом деле их можно извлечь с помощью курсора Фактически, результирующее множество, возвращаемое командой **OPEN,** было предопределено в момент открытия курсора. Мы не прочитали при открытии курсора ни одного блока данных таблицы, но результат оказался жестко зафиксированным. Мы не сможем узнать этот результат, пока не извлечем данные, но с точки зрения нашего курсора результат этот неизменен. Дело не в том, что СУБД Oracle скопировала все эти данные в другое место при открытии курсора; данные сохранил оператор **delete,** поместив их в область данных под названием сегмент отката.

Именно в этом и состоит согласованность по чтению, и если не понимать, как работает схема многовариантности в Oracle и каковы ее последствия, вы не только не сможете воспользоваться всеми преимуществами СУБД Oracle, но и не создадите корректных приложений для Oracle, гарантирующих целостность данных.

Давайте рассмотрим последствия использования многовариантной согласованности запросов по чтению и неблокируемых чтений. Для тех, кто не знаком с многовариантностью, представленные ниже результаты могут показаться удивительными. Для простоты предположим, что в каждом блоке данных (это минимальная единица хранения в СУБД) считываемой таблицы хранится всего одна строка и что выполняется полный просмотр таблицы.

В запрашиваемой таблице хранятся балансы банковских счетов. Она имеет очень простую структуру:

```
create table accounts
( account_number number primary key,
  account_balance number
);
```

В реальной таблице счетов будут сотни тысяч строк, но для простоты мы будем рассматривать таблицу всего с четырьмя строками (более детально мы рассмотрим этот пример в главе 3):

Строка	***Номер счета***	*Баланс* ***счета***
1	123	500.00 $
2	234	250,00 $
3	345	400,00 $
4	456	100,00 $

Требуется создать отчет, который в конце банковского дня позволяет определить количество денег в банке. Это делается с помощью предельно простого запроса:

```
select sum(account balance) from accounts;
```

Конечно, в данном примере ответ очевиден — 1250 $. Однако что произойдет, если мы прочитаем строку I, а при считывании строк 2 и 3 с одного из банкоматов будет вы-

полнена транзакция, переводящая 400 $ со счета 123 на счет 456? Наш запрос прочтет 500 $ в строке 4 и выдаст результат 1650 $, не так ли? Конечно, этого надо избежать, так как подобный результат ошибочен — никогда такого баланса по счетам в базе данных не было. Нужно понять, как СУБД Oracle избегает подобных ситуаций и чем отличаются используемые при этом методы от используемых во всех остальных СУБД.

Практически в любой другой СУБД для получения "согласованного" и "корректного" ответа на этот запрос необходимо блокировать либо всю таблицу, по которой идет суммирование, либо строки по мере их чтения. Это предотвратит изменение результата другими сеансами в ходе его получения. Если заблокировать всю таблицу, будет получен результат, соответствующий состоянию базы данных в момент начала выполнения запроса. Если блокировать данные по мере чтения (такая разделяемая блокировка чтения предотвращает изменения, но не чтение данных другими сеансами), будет получен результат, соответствующий состоянию базы данных в момент завершения выполнения запроса. Оба эти метода существенно снижают возможности одновременного доступа. Блокировка таблицы предотвращает любые изменения таблицы во время выполнения запроса (для таблицы из четырех строк этот период очень короток, но для таблиц с сотнями тысяч строк запрос может выполняться несколько минут). Метод "блокирования по ходу чтения" предотвращает изменение уже прочитанных и обработанных данных и потенциально может приводить к взаимным блокировкам выполнения вашего запроса и других изменений.

Как уже было сказано, вы не сможете в полном объеме использовать преимущества СУБД Oracle, если не понимаете концепцию многовариантности. В СУБД Oracle многовариантность используется для получения результатов, соответствующих моменту начала выполнения запроса, при этом не блокируется ни единой строки (пока транзакция по переводу денег изменяет строки 1 и 4, они будут заблокированы от других изменений, но не от чтения, выполняемого, например, нашим запросом SELECT SUM...)- Фактически в СУБД Oracle нет "разделяемых блокировок чтения", типичных для других СУБД, — они в ней просто не нужны. Все устранимые препятствия для одновременного доступа были устранены.

Итак, как же СУБД Oracle получает корректный, согласованный результат (1250 $) при чтении, не блокируя данных, другими словами, не мешая одновременному доступу? Секрет — в механизме выполнения транзакций, используемом в СУБД Oracle. При любом изменении данных Oracle создает записи в двух разных местах. Одна запись попадает в журналы повторного выполнения, где Oracle хранит информацию, достаточную для повторного выполнения, или "наката", транзакции. Для оператора вставки это будет вставляемая строка. Для оператора удаления это будет запрос на удаление строки в слоте X блока Y файла Z. И так далее. Другая запись — это запись отмены, помещаемая в сегмент отката. Если транзакция завершается неудачно и должна быть отменена, СУБД Oracle будет читать "предварительный" образ из сегмента отката, восстанавливая необходимые данные. Помимо отмены транзакций, СУБД Oracle использует сегменты отката для отмены изменений в блоках при их чтении, то есть для восстановления данных блока на момент начала выполнения запроса. Это позволяет читать данные несмотря на блокировку и получать корректные, согласованные результаты, не блокируя данные.

Итак, в нашем примере Oracle получает результат следующим образом:

Время	*Запрос*	*Транзакция по переводу со счета на счет*
T1	Читает строку 1, sum получает значение 500 $	
T2		Изменяет строку 1, устанавливает исключительную блокировку на строку 1, предотвращая другие изменения. В строке 1 теперь хранится значение 100 $
T3	Читает строку 2, sum получает значение 750 $	
T4	Читает строку 3, sum получает значение 1150 $	
T5		Изменяет строку 4, устанавливает исключительную блокировку на строку 4, предотвращая другие изменения (но не чтение). В строке 4 теперь хранится значение 500 $.
T6	Читает строку 4, определяет, что она была изменена. Выполняется откат блока до того состояния, которое он имел в момент времени T1. Запрос затем прочитает значение 100 $ из этого блока	
T7		Транзакция фиксируется
T8	Выдает 1250 $ в качестве результата суммирования	

В момент времени T6 СУБД Oracle фактически "читает поверх" блокировки, установленной транзакцией на строке 4. Именно так реализуется неблокируемое чтение: СУБД Oracle просто проверяет, изменились ли данные, игнорируя тот факт, что они в настоящий момент заблокированы (т.е. определенно изменены). Она извлечет старое значение из сегмента отката и перейдет к следующему блоку данных.

Это еще одна убедительная демонстрация многовариантности: в базе данных имеется несколько версий одной и той же информации, по состоянию на различные моменты времени. СУБД Oracle использует эти сделанные в разное время "моментальные снимки" данных для поддержки согласованности по чтению и неблокируемости запросов.

Это согласованное по чтению представление данных всегда выполняется на уровне оператора SQL, — результаты выполнения любого оператора SQL всегда согласованы на момент его начала. Именно это свойство позволяет получать предсказуемый набор данных в результате, например, следующих вставок:

```
for x in (select * from t)
loop
```

```
    insert into t values (x.username, x.user_id, x.created);
end loop;
```

Результат выполнения оператора **SELECT * FROM** T предопределен в момент начала выполнения запроса. Оператор **SELECT не** будет "видеть" новых данных, генерируемых операторами **INSERT.** Представьте себе, что было бы в противном случае: оператор превратился бы в бесконечный цикл. Если бы по мере генерации оператором INSERT дополнительных строк в таблице T, оператор SELECT мог "видеть" эти вставляемые строки, представленный выше фрагмент кода создал бы неизвестное количество строк. Если бы в таблице T первоначально было 10 строк, в результате могло бы получиться 20, 21, 23 или бесконечное количество строк. Точно предсказать результат было бы невозможно. Согласованность по чтению обеспечивается для всех операторов, так что операторы **INSERT,** вроде представленного ниже, тоже работают предсказуемо:

```
insert into t select * from t;
```

Оператор INSERT получит согласованное по чтению представление таблицы T — он не "увидит" строки, которые сам же только что вставил, и будет вставлять только строки, существовавшие на момент начала его выполнения. Во многих СУБД подобные рекурсивные операторы просто не разрешены, поскольку они не могут определить, сколько строк вообще будет вставлено.

Поэтому если вы привыкли к реализации согласованности и одновременности запросов в других СУБД или просто никогда не сталкивались с такими понятиями (не имеете реального опыта работы с СУБД), то теперь понимаете, насколько важно для вашей работы их понимание. Чтобы максимально использовать потенциальные возможности СУБД Oracle, необходимо понимать эти проблемы и способы их решения именно в Oracle, а не в других СУБД.

Независимость от СУБД?

Вы, наверное, уже поняли направление моей мысли. Я ссылался на другие СУБД и описывал различия реализации одних и тех же возможностей в каждой из них. Я убежден: за исключением некоторых приложений, исключительно читающих из базы данных, создать полностью независимое от СУБД и при этом масштабируемое приложение крайне сложно и даже практически невозможно, не зная особенностей работы всех СУБД.

Например, давайте вернемся к первому примеру планировщика ресурсов (до добавления конструкции **FOR UPDATE).** Предположим, это приложение было разработано на СУБД с моделью блокирования/обеспечения одновременного доступа, полностью отличающейся от принятой в Oracle. Я собираюсь продемонстрировать, что при переводе приложения с одной СУБД на другую необходимо проверять, работает ли оно корректно в новой среде.

Предположим, что первоначально приложение по планированию ресурсов работало в СУБД, использующей блокирование на уровне страниц и блокировку чтения (чтение блокируется при изменении считываемых данных), и для таблицы **SCHEDULES** был создан индекс:

```
create index schedules_idx on schedules(resource name, start_time);
```

Предположим также, что бизнес-правило было реализовано с помощью триггера (после выполнения оператора **INSERT,** но перед фиксацией транзакции мы проверяем, что для указанного временного интервала в базе данных имеется только наша, только что вставленная строка). В системе с блокированием на уровне страниц, из-за изменения страницы индекса по столбцам **RESOURCE_NAME** и **START_TIME,** очень вероятно, что транзакции будут выполняться строго последовательно. Система будет выполнять вставки поочередно, поскольку страница индекса блокируется (все близкие значения по полю START_TIME для одного ресурса **RESOURCE_NAME** будут находиться на той же странице). В такой СУБД с блокированием на уровне страниц наше приложение, вероятно, будет работать нормально, так как перекрытие выделяемых ресурсов будет проверяться последовательно, а не одновременно.

Если просто перенести это приложение в СУБД Oracle, исходя из предположения, что она работает точно так же, можно получить шок. В СУБД Oracle, выполняющей блокирование на уровне строк и не блокирующей чтения, оно окажется некорректным. Как уже было показано, необходимо использовать конструкцию FOR UPDATE для упорядочения доступа. Без этой конструкции два пользователя могут зарезервировать ресурс на одно и то же время. Это будет прямым следствием непонимания особенностей работы используемой СУБД в многопользовательской среде.

С подобными проблемами я сталкивался многократно при переносе приложений из СУБД А в СУБД Б. Когда приложение, без проблем работавшее в СУБД А, не работает или работает весьма странно в СУБД Б, сразу же возникает мысль, что "СУБД Б — плохая". Правда, однако, в том, что СУБД Б работает **иначе.** Ни одна из СУБД не ошибается и не является "плохой" — они просто разные. Знание и понимание особенностей их работы поможет успешно решить подобные проблемы.

Совсем недавно я помогал перевести код с языка Transact SQL (язык создания хранимых процедур для СУБД SQL Server) на PL/SQL. Разработчик, занимавшийся переводом, жаловался, что SQL-запросы в Oracle возвращают "неправильный" ответ. Запросы выглядели следующим образом:

```
declare
     l_soma_variable    varchar2(25);
begin
    if  (sorae_condition)
    then
         l_some_variable  : = f (...);
    end  if;

    for  x  in  (select  *  from  T  where  x  =  l_some_variable)
    loop
```

Целью является получение всех строк таблицы T, которые в столбце X имеют пустое значение, если некоторое условие не выполнено, или определенное значение, если это условие выполнено.

Суть жалобы состояла в том, что, в Oracle этот запрос не возвращал данных, если **переменная L_SOME_VARIABLE** не получала значения явно (когда у нее оставалось

значение NULL). В СУБД Sybase или SQL Server все было не так — запрос находил строки с пустым **(NULL)** значением в столбце X. Я встречался с этим практически при любом переводе приложения с СУБД Sybase или SQL Server на Oracle. Язык SQL предполагает использование трехзначной логики, и СУБД Oracle реализует пустые значения так, как того требует стандарт ANSI SQL. По этим правилам сравнение столбца X со значением NULL не дает ни **True,** ни **False** — результат фактически **неизвестен.** Следующий пример показывает, что я имею в виду:

```
ops$tkyte@ORA8I.WORLD> select * from dual;

D

X

ops$tkyte@ORA8I.WORLD> select * from dual where null=null;

no rows selected

ops$tkyte@ORA8I.WORLD> select * from dual where null<>null;

no rows selected
```

В первый раз это может показаться странным: в Oracle NULL не равен и не не равен NULL. СУБД SQL Server по умолчанию ведет себя не так: в SQL Server и Sybase NULL равен NULL. Ни Oracle, ни Sybase, ни SQL Server не выполняет операторы SQL **неправильно** — они просто делают это **по-разному.** Все эти СУБД якобы соответствуют стандарту ANSI, но все равно работают по-разному. Есть неоднозначности, проблемы совместимости с прежними версиями и так далее, которые необходимо решать. Например, СУБД SQL Server поддерживает метод сравнения со значением **NULL,** диктуемый стандартом ANSI, но не по умолчанию (это нарушило бы работу тысяч уже существующих приложений, созданных для этой СУБД).

Одним из решений проблемы могло быть переформулирование запроса следующим образом:

```
select *
  from t
  where (x = l_some_variable OR (x is null and l_some_variable is NULL))
```

Однако это привело бы к еще одной проблеме. В СУБД SQL Server при выполнении этого запроса использовался бы индекс по столбцу X. В СУБД Oracle индекс на основе B*-дерева (подробнее о методах индексирования читайте в главе 7) не позволяет индексировать значения ключа NULL. Поэтому, если необходимо найти пустые значения, индексы на основе B*-деревьев не сильно помогут.

В рассматриваемом случае, чтобы свести к минимуму изменения в коде, столбцу X присваивалось значение, которого не могло быть в реальных данных. Так, X, по определению, был числом положительным, поэтому было выбрано значение -1. Запрос приобрел следующий вид:

```
select * from t where nvl(x,-l) = nvl(l_some_variable,-1)
```

Мы создали индекс по функции:

```
create index t_idx on t(nvl(x,-1));
```

С минимальными изменениями мы добились того же результата. Отсюда можно сделать следующие важные выводы.

- СУБД — различны. Опыт работы с одной может оказаться полезен в другой, но нужно быть готовым к ряду **принципиальных** отличий и многим очень мелким.
- Мелкие различия (вроде обработки NULL-значений) могут иметь такое же влияние, как и принципиальные (например, механизм управления одновременным доступом).
- Единственный способ справиться с этими проблемами — знать особенности работы СУБД и уметь реализовать предоставляемые ею возможности.

Разработчики часто спрашивают меня, как сделать в СУБД что-то конкретное. Например, меня спрашивают: "Как создать временную таблицу в хранимой процедуре?". На такие вопросы я не даю прямого ответа — я всегда отвечаю вопросом: "А для чего вам это нужно?". Неоднократно в ответ я слышал: "Мы создавали временные таблицы в хранимых процедурах в SQL Server, и теперь нам надо это сделать в Oracle". Именно это я и предполагал услышать. В таком случае мой ответ прост: "Вы ошибаетесь, думая, что надо создавать временные таблицы в хранимой процедуре в Oracle". На самом деле в СУБД Oracle это будет крайне неудачным решением. При создании таблиц в хранимых процедурах в Oracle вскоре обнаружится, что:

- выполнение операторов ЯОД в этом контексте снижает масштабируемость;
- постоянное выполнение операторов ЯОД снижает производительность;
- выполнение операторов ЯОД приводит к фиксации транзакции;
- для доступа к этой таблице во всех хранимых процедурах придется использовать динамический SQL, т.к. статический SQL использовать невозможно;
- динамический SQL в PL/SQL оптимизируется хуже и работает медленнее статического.

Итак, не надо делать в точности так, как в SQL Server (если временная таблица в Oracle вообще понадобится). Делать следует то, что является наиболее оптимальным для Oracle. При обратном переходе из Oracle в SQL Server тоже не стоит создавать одну большую таблицу с временными данными для всех пользователей (как это делается в Oracle). Это приведет к снижению масштабируемости и возможностей одновременного доступа в данной СУБД. Каждая СУБД имеет существенные отличия.

Влияние стандартов

Если все СУБД соответствуют стандарту SQL92, они должны быть одинаковы. Так считают многие. Сейчас я развею этот миф.

SQL92 — это стандарт ANSI/ISO для СУБД. Он является развитием стандарта ANSI/ISO SQL89. Этот стандарт задает язык (SQL) и поведение (транзакции, уровни изоли-

рованности и т.д.) для СУБД. Знаете ли вы, что многие коммерческие СУБД соответствуют стандарту SQL92? А знаете ли, как немного это значит для переносимости запросов и приложений?

Начиная читать стандарт SQL92, обнаруживаешь, что он имеет четыре уровня.

- **Начальный.** Именно этому уровню соответствует большинство предлагаемых СУБД. Этот уровень является незначительным развитием предыдущего стандарта, SQL89. Ни одна СУБД не сертифицирована по более высокому уровню. Более того, фактически Национальный институт стандартов и технологий (National Institute of Standards and Technology — **NIST),** агенство, сертифицировавшее соответствие стандартам SQL, сертификацией больше не занимается. Я входил в состав команды, сертифицировавшей Oracle 7.0 в NIST как соответствующий начальному уровню стандарта SQL92 в 1993 году. СУБД, соответствующая начальному уровню этого стандарта, поддерживает набор возможностей Oracle 7.0.
- **Переходный.** С точки зрения поддерживаемых возможностей это что-то среднее между начальным и промежуточным уровнем.
- **Промежуточный.** Этот уровень добавляет много возможностей, в том числе (этот список далеко не исчерпывающий):
  - динамический SQL;
  - каскадное удаление для обеспечения целостности ссылок;
  - типы данных DATE и TIME;
  - домены;
  - символьные строки переменной длины;
  - выражения **CASE;**
  - функции **CAST** для преобразования типов данных.
- **Полный.** Добавляет следующие возможности (этот список тоже не исчерпывающий):

  управление подключением;

  тип данных **BIT** для битовых строк;

  отложенная проверка ограничений целостности;

  производные таблицы в конструкции **FROM;**

  подзапросы в конструкции **CHECK;**

  временные таблицы.

В стандарт начального уровня не входят такие конструкции, как внешние соединения, новый синтаксис для внутренних соединений и т.д. Переходный уровень требует поддержки соответствующего синтаксиса внешнего и внутреннего соединения. Промежуточный уровень добавляет новые возможности, а полный и представляет собой, собственно, SQL92. В большинстве книг по SQL92 не различаются эти уровни поддержки, что сбивает с толку. В них демонстрируется, как должна работать "идеальная" СУБД,

полностью реализующая стандарт SQL92. В результате нельзя взять книгу по SQL92 и применить представленные в ней приемы к СУБД, соответствующей стандарту SQL92. Например, в СУБД SQL Server предлагаемый стандартом синтаксис "внутреннего соединения" в SQL-операторах поддерживается, а в СУБД Oracle — нет. Но обе эти СУБД соответствуют стандарту SQL92. В СУБД Oracle можно выполнять внешние и внутренние соединения, но делать это надо не так, как в SQL Server. В результате начальный уровень стандарта SQL92 мало что дает, а при использовании средств более высоких уровней возможны проблемы при переносе на другую СУБД.

Не надо бояться использовать специфические средства конкретной СУБД, — за них заплачено немало денег. В каждой СУБД есть свой набор уникальных возможностей, и в любой СУБД можно найти способ выполнить необходимое действие. Используйте в текущей СУБД лучшее и реализуйте новые компоненты при переходе на другие СУБД. Используйте соответствующие приемы программирования, максимально изолирующие остальную часть приложения от этих изменений. Эти же приемы программирования применяются разработчиками переносимых приложений, поддерживающих несколько ОС. Цель в том, чтобы в полной мере использовать имеющиеся средства, но при этом иметь возможность менять реализацию в каждом конкретном случае.

Например, типичная функция многих приложений баз данных — генерация уникального ключа для каждой строки. При вставке строки система должна автоматически сгенерировать ключ. В Oracle для этого предлагается объект базы данных — последовательность (SEQUENCE). В Informix имеется тип данных SERIAL. Sybase и SQL Server поддерживают тип данных IDENTITY. В каждой СУБД имеется способ решить эту задачу. Однако методы решения различны, различны и возможные последствия их применения. Поэтому знающий разработчик может выбрать один из двух вариантов:

- разработать метод генерации уникального ключа, полностью независимый от СУБД;
- согласиться с разными реализациями и использовать разные методы генерации ключей в зависимости от СУБД.

Теоретическое преимущество первого подхода состоит в том, что при переходе с одной СУБД на другую ничего менять не придется. Я назвал это преимущество "теоретическим", поскольку недостатки такого решения настолько велики, что делают его практически неприемлемым. Для создания полностью независимого от СУБД процесса придется создать таблицу вида:

```
create table id_table (id_name varchar(30), id_value number);
insert into id_table values ('MY_KEY', 0);
```

Затем для получения нового ключа необходимо выполнить следующий код:

```
update id_table set id_value = id_value + 1 where id_name = 'MY_KEY';
select id_value from id_table where id_name = 'MY_KEY';
```

Выглядит он весьма просто, но выполнять подобную транзакцию в каждый момент времени может только один пользователь. Необходимо изменить соответствующую строку, чтобы увеличить значение счетчика, а это приведет к поочередному выполнению операций. Не более одного сеанса в каждый момент времени будет генерировать новое зна-

чение ключа. Проблема осложняется тем, что реальные транзакции намного больше транзакции, показанной выше. Показанные в примере операторы UPDATE и SELECT — лишь два из множества операторов, входящих в транзакцию. Необходимо еще вставить в таблицу строку с только что сгенерированным ключом и выполнить необходимые действия для завершения транзакции. Это упорядочение доступа будет огромным ограничивающим фактором для масштабирования. Подумайте о последствиях, если этот метод применить для генерации номеров заказов в приложении для обработки заказов на Web-сайте. Одновременная работа нескольких пользователей станет невозможной, — заказы будут обрабатываться последовательно.

Правильное решение этой проблемы состоит в использовании для каждой СУБД соответствующего кода. В Oracle (предполагается, что уникальный ключ необходимо генерировать для таблицы T) лучшим способом будет:

```
create table t (pk number primary key, . . . ) ;
create sequence t_seq;
create trigger t_trigger before insert on t for each row
begin
    select t_seq.nextval into :new.pk from dual;
end;
```

В результате каждая вставляемая строка автоматически и незаметно для приложения получит уникальный ключ. Тот же эффект можно получить и в других СУБД с помощью их типов данных — синтаксис оператора создания таблицы изменится, а результат будет тем же. Мы использовали второй вариант — специфические средства каждой СУБД для **неблокируемой,** высокопараллельной генерации уникального ключа, что, однако, не потребовало реальных изменений в коде приложения — все необходимые действия выполнены операторами ЯОД.

Приведу еще один пример безопасного программирования, обеспечивающего переносимость. Если понятно, что **каждая СУБД реализует одни и те же возможности поразному,** можно при необходимости создать дополнительный уровень доступа к базе данных. Предположим, вы программируете с использованием интерфейса JDBC. Если используются только простые операторы **SQL, SELECT, INSERT, UPDATE** и **DELETE,** дополнительный уровень абстракции скорее всего не нужен. Можно включать код SQL непосредственно в приложение, если использовать конструкции, поддерживаемые во всех СУБД, с которыми должно работать приложение. Другой подход, одновременно упрощающий перенос и повышающий производительность, состоит в использовании хранимых процедур, возвращающих результирующие множества. Если разобраться, окажется, что все СУБД могут возвращать результирующие множества из хранимых процедур, но способы при этом используются абсолютно разные. Для каждой СУБД придется написать свой исходный код.

Теперь появляется выбор — либо не использовать хранимые процедуры, возвращающие результирующие множества, либо писать отдельный исходный код для каждой СУБД. Я, несомненно, выбрал бы метод "отдельный код для каждой СУБД" и активно использовал бы хранимые процедуры. Казалось бы, что при этом для перехода на другую СУБД потребуется больше времени. Однако оказывается, что этот подход упрощает создание приложений, переносимых на различные СУБД. Вместо поисков идеально-

го кода SQL, работающего **во всех** СУБД (причем, как правило, в одних лучше, а в других — хуже), используется код SQL, максимально эффективный в конкретной СУБД. Ею можно вынести из приложения, что дает дополнительные возможности настройки. Можно исправить запрос с низкой производительностью непосредственно в СУБД, и это изменение будет немедленно учтено, без исправлений в приложении. Кроме того, применяя этот метод, можно свободно и в полном объеме использовать преимущества предлагаемых производителем СУБД расширений языка SQL. Например, СУБД Oracle поддерживает иерархические запросы с помощью конструкции **CONNECT** BY в операторах SQL. Эта уникальная возможность очень поможет при создании рекурсивных запросов. В Oracle вы свободно сможете использовать это расширение SQL, поскольку оно — "вне" приложения (скрыто в базе данных). В других СУБД для достижения аналогичных результатов, возможно, придется использовать временные таблицы и хранимые процедуры. Вы заплатили за эти возможности, так почему же их не использовать.

Такие же методы используют разработчики, создавая код, предназначенный для работы на множестве платформ. Корпорация Oracle, например, применяет описанную выше методику при разработке СУБД. Есть большой фрагмент кода (составляющий, однако, небольшую часть всего кода СУБД), который называется OSD-код (Operating System Dependent) и создается отдельно для каждой платформы. С помощью этого уровня абстракции в СУБД Oracle можно использовать специфические возможности ОС для обеспечения высокой производительности и интегрирования, не переписывая при этом код самой СУБД. Именно благодаря этому СУБД Oracle может работать как многопотоковое приложение в Windows и как многопроцессное — в UNIX. Механизмы межпроцессного взаимодействия абстрагированы до такого уровня, что могут воплощаться по-разному для каждой ОС; при этом обеспечивается такая же производительность, как и в приложениях, написанных специально для данной платформы.

Помимо синтаксических различий в языке SQL, различаются реализации операторов, различной будет и производительность выполнения одного и того же запроса, есть проблемы управления одновременным доступом, уровней изолированности транзакций, согласованности запросов и т.д. Все это более детально будет рассмотрено в главах 3 и 4, — мы увидим, как сказываются эти различия. В стандарте SQL92 попытались дать четкие определения того, как должна выполняться транзакция, как должны обеспечиваться уровни изолированности, но в конечном итоге в разных СУБД результаты получаются различными. Все это связано с реализацией. В одной СУБД приложение будет вызывать взаимные блокировки и заблокирует все, что можно. В другой СУБД это же приложение не вызывает никаких проблем и работает отлично. В одной СУБД блокирование (физически упорядочивающее обращения) намеренно использовалось в приложении, а при его переносе в другую СУБД, где блокирования нет, получается неверный ответ. Чтобы перенести готовое приложение в другую СУБД, требуется много труда и усилий, даже если при первоначальной разработке неукоснительно соблюдался стандарт.

Возможности и функции

Противники обязательного обеспечения "независимости от СУБД" приводят следующий аргумент: нужно хорошо понимать, что именно предлагает конкретная СУБД, и полностью использовать ее возможности. В этом разделе не описываются все уникаль-

ные возможности Oracle 8i, — для этого понадобилась бы отдельная большая книга. Новым возможностям СУБД Oracle 8i посвящена специальная книга в наборе документации по СУБД Oracle. Если учесть, что вместе с СУБД Oracle поставляется документация общим объемом около 10 000 страниц, детальное рассмотрение каждой возможности и функции практически нереально. В этом разделе просто показано, почему даже поверхностное представление об имеющихся возможностях дает огромные преимущества.

Как уже было сказано, я отвечаю на вопросы о СУБД Oracle на Web-сайте. Если честно, процентов 80 моих ответов — ссылки (URL) на документацию. Меня спрашивают, как реализовать те или иные сложные функциональные возможности в базе данных (или вне ее). А я просто даю ссылку на соответствующее место в документации, где написано, как это уже реализовано в СУБД Oracle и как этими возможностями пользоваться. Часто такие случаи бывают с репликацией. Я получаю вопрос: "Хотелось бы сохранять копию данных в другом месте. Эта копия должна быть доступна только для чтения. Обновление должно выполняться раз в сутки, в полночь. Как написать соответствующий код?". Ответ простой: см. описание оператора CREATE SNAPSHOT. Вот что такое встроенные возможности СУБД.

Можно, конечно, для интереса написать собственный механизм репликации, но это будет не самое разумное действие. СУБД делает многое и, как правило, лучше, чем создаваемые нами приложения. Репликация, например, встроена в ядро, написанное на языке C. Она работает быстро, сравнительно проста в использовании и надежна. Работает в разных версиях, на разных платформах. При возникновении проблем служба поддержки Oracle поможет их решить. После обновления версии репликация будет поддерживаться с новыми, дополнительными возможностями. Теперь предположим, что вы разработали собственный механизм репликации. Вам придется заняться его поддержкой во всех версиях СУБД, которые вы собираетесь поддерживать. Одинаковое функционирование в версии 7.3, 8.0, 8.1 и 9.0 и так далее вы должны будете обеспечивать сами. Если произойдет сбой, обращаться будет не к кому. По крайней мере, пока не удастся получить маленький тестовый пример, демонстрирующий основную проблему. При выходе новой версии вам придется самостоятельно переносить в нее код механизма репликации.

Недостаточное понимание того, что предлагает СУБД, может серьезно помешать в будущем. Недавно разработчики демонстрировали мне созданное ими "очень нужное" программное обеспечение. Это была система обмена сообщениями, решавшая проблему очередей в базе данных. Она обычно возникает при необходимости использования таблицы несколькими сеансами в качестве "очереди". Необходимо, чтобы несколько пользователей могли заблокировать очередную запись, пропустив все уже заблокированные записи (они уже обрабатываются). Проблема в том, что нет задокументированной возможности СУБД для пропуска заблокированных строк. Поэтому, не зная о существовании средств, предоставляемых СУБД Oracle, можно приняться за реализацию поддержки очередей самостоятельно (или приобрести готовое решение).

Именно это и сделала упомянутая группа разработчиков. Они создали набор процессов и придумали функциональный интерфейс для организации очередей сообщений в СУБД. Они потратили на это немало времени и сил и были уверены, что сделали нечто

действительно уникальное. Когда я увидел систему в действии и узнал ее функциональные возможности, мне осталось сказать лишь одно: это аналог расширенной поддержки очередей, Advanced Queues. Эта возможность давно встроена в СУБД. Она решает задачу "получить первую незаблокированную запись в очереди и заблокировать ее". Все, что нужно, уже сделано. Разработчики, не зная о существовании такой возможности, потратили на ее реализацию много времени и сил. Кроме того, им придется тратить немало времени и на ее поддержку в дальнейшем. Их руководитель не очень обрадовался, узнав, что вместо уникального программного обеспечения получилась эмуляция встроенной возможности СУБД.

Я видел, как разработчики в СУБД Oracle 8i создавали процессы-демоны, читающие сообщения из программных каналов (это механизм межпроцессного взаимодействия в СУБД). Процессы-демоны выполняли операторы SQL, содержавшиеся в прочитанных из программного канала сообщениях, и фиксировали сделанное. Это делалось для проверки транзакций, чтобы записи проверки откатывались при откате основной транзакции. Обычно если для проверки доступа к данным использовались триггеры и основной оператор впоследствии выполнить не удавалось, все изменения откатывались (см. главу 4, где неделимость операторов рассматривается более детально). Посылая же сообщение другому процессу, можно записывать информацию в другой транзакции и фиксировать ее независимо. Запись проверки при этом оставалась, даже если основная транзакция откатывалась. В версиях Oracle до Oracle 8i это был приемлемый (и практически единственный) способ реализации описанной функции. Когда я рассказал разработчикам об автономных транзакциях, поддерживаемых СУБД (мы их подробно рассмотрим в главе 15), они очень расстроились. Автономные транзакции, реализуемые добавлением единственной строки кода, делали то же, что вся их система. Положительным моментом оказалось то, что можно было выкинуть существенную часть кода и не поддерживать его в дальнейшем. Кроме того, система заработала быстрее и стала проще для понимания. Но их это все равно мало радовало, — очень уж много времени было потрачено на изобретение велосипеда. Особенно расстроился создатель процессов-демонов, плоды трудов которого были отправлены в мусорную корзину.

С подобными случаями я сталкиваюсь постоянно: затрачиваются громадные усилия на решение проблем, уже давно решенных в самой СУБД. Если вы не потратите время на изучение того, что предлагается, рано или поздно будете наказаны, изобретая велосипед. Во второй части книги, "Структуры и утилиты базы данных", мы детально рассмотрим отдельные функциональные возможности, предлагаемые СУБД. Я выбрал те возможности и функции, которые часто используют разработчики или которые стоило бы использовать намного чаще. Описана будет, однако, лишь вершина айсберга. В СУБД Oracle намного больше средств и возможностей, чем можно описать в одной книге.

Решайте проблемы просто

Всегда есть два способа решения любой проблемы: простой и сложный. Но люди почему-то всегда выбирают сложный. Это не всегда делается намеренно, чаще — по незнанию. Разработчики просто не предполагают, что СУБД может делать "это". Я же предполагаю, что СУБД может делать все, и пишу что-то собственноручно, только если оказывается, что этого она не делает.

Например, меня часто спрашивают: "Как сделать, чтобы пользователь мог подключиться к базе данных только один раз?". (Есть еще сотня примеров, которые я мог бы здесь привести в качестве иллюстрации.) Наверное, это требование многих приложений; правда, в моей практике разработки такие приложения не встречались — я не вижу веской причины для того, чтобы ограничивать пользователей подобным образом. Однако другим разработчикам это нужно, и они обычно придумывают сложное решение. Например, создают пакетное задание, выполняемое операционной системой и просматривающее представление V$SESSION, а затем произвольно прекращающее сеансы пользователей, подключившихся к базе данных более одного раза. Или создают собственные таблицы, в которые приложение вставляет строку при регистрации пользователя и удаляет ее по завершении работы. Подобная реализация неизбежно приводит к многочисленным обращениям в службу поддержки, поскольку если приложение завершает работу нештатно, строка из этой таблицы не удаляется. Я видел еще много "творческих" способов добиться этого, но ни один из них не был таким простым:

```
ops$tkyte@0RA8I.W0RLD> create profile one_session limit sessions_per_user 1;
Profile created.

ops$tkyte@ORA8l.WORLD> alter user scott profile one_session;
User altered.

ops$tkyte@ORA8I.WORLD> alter system set resource_limit=true;
System altered.
```

Вот и все. Теперь любой пользователь с профилем ONE_SESSION может подключиться только один раз. Простота этого решения обычно приводит разработчиков в восторг и вызывает запоздалые сожаления. Потратьте время на ознакомление с имеющимися средствами и их возможностями — это позволит сэкономить много времени и сил при разработке.

Тот же принцип "делай проще" применяется и на более высоком, архитектурном уровне. Я рекомендую подумать дважды, прежде чем браться за сложные реализации. Чем больше "движущихся частей" в системе, тем больше компонентов, которые могут работать неверно, а при использовании сложной архитектуры определить, что именно является причиной ошибки, будет непросто. Может быть, использование "надцатиуровневой" архитектуры — это действительно "круто", но лишено смысла, если в простой хранимой процедуре можно сделать то же самое, но лучше, быстрее и с использованием меньших ресурсов.

Я участвовал в разработке приложения, продолжающейся более года. Это было Web-приложение, используемое в масштабе компании. Клиент на базе HTML и с использованием технологии JSP динамически получал страницы с сервера промежуточного уровня, который взаимодействовал с CORBA-объектами, в свою очередь, обращавшимися к СУБД. CORBA-объекты должны были поддерживать "состояние" и подключаться к СУБД для организации сеанса. В ходе тестирования этой системы оказалось, что потребуется много серверов приложений и очень мощная машина для работы СУБД, чтобы поддерживать 10000, как предполагалось, одновременно работающих пользователей. Более того, иногда возникала проблема нестабильности, связанная со сложностью взаимодействия

компонентов (ответить на вопрос, где именно и почему произошла ошибка в этой сложной системе, было трудно). Система масштабировалась, но требовала при этом огромных ресурсов. Кроме того, поскольку для реализации использовалось много сложных технологий, для разработки и сопровождения системы требовалось много опытных программистов. Мы разобрались в этой системе и ее предполагаемых функциях и поняли, что архитектура ее несколько сложнее, чем необходимо для решения поставленных задач. Мы увидели, что с помощью модуля **PL/SQL** сервера приложений Oracle iAS и ряда хранимых процедур можно было сделать такую же систему, работающую на существенно менее мощном оборудовании, причем усилиями менее опытных разработчиков. Никаких компонентов EJB, никаких сложных взаимодействий между страницами JSP и компонентами EJB — обычное преобразование указанного адреса URL в вызов хранимой процедуры. Эта новая система работает и используется до сих пор, поддерживает больше пользователей, чем предполагалось, и работает так быстро, что порой не верится. Она использует самую простую архитектуру, минимум компонентов, работает на дешевом 4-процессорном сервере уровня рабочих групп и никогда не дает сбоев (ну, один раз табличное пространство переполнилось, но это уже другая проблема).

Для решения задачи я всегда предпочитаю наиболее простую архитектуру. Результат часто получается потрясающий. Для каждой технологии есть соответствующие инструменты — не всегда надо просто гвоздь забить, так что может понадобиться что-то кроме молотка...

Открытость

Есть еще одна причина, почему при разработке часто выбирается сложный способ решения проблемы, — сложившееся представление, что надо жертвовать всем ради "открытости" и "независимости от СУБД". Разработчики хотят избежать использования "закрытых", "специфических" возможностей СУБД — иногда даже таких простых, как хранимые процедуры или последовательности, поскольку это привяжет их к определенной СУБД. Я настаиваю на том, что если создается приложение, читающее и **изменяющее** данные, оно уже в некоторой степени привязано к СУБД. Когда запросы начнут выполняться одновременно с изменениями, вы сразу обнаружите небольшие (а иногда — и большие) отличия в работе СУБД. Например, в одной СУБД может оказаться, что оператор **SELECT COUNT(*) FROM** T вступает во взаимную блокировку с простым изменением двух строк. В Oracle же запрос **SELECT COUNT(*)** никогда не блокирует другие сеансы. Мы уже рассматривали пример, когда в одной СУБД бизнес-правило работало как побочный эффект используемой модели блокирования, а в другой СУБД — нет. Было показано, что при одном и том же порядке выполнения транзакций в различных СУБД приложение может давать разные результаты. Причина — принципиальные различия в реализациях. Вы со временем поймете, что лишь очень немногие приложения можно непосредственно перенести из одной в другую СУБД. Различия в интерпретации (например, выражения **NULL=NULL)** и обработке операторов **SQL** будут всегда.

В одном из недавних проектов разработчики создавали Web-приложение с использованием Visual Basic, управляющих элементов ActiveX, Web-сервера **IIS** и СУБД Oracle 8i. Разработчики выразили опасение по поводу реализации бизнес-логики на языке PL/SQL — приложение становится зависимым от СУБД — и спрашивали, можно ли это исправить.

Меня этот вопрос несколько удивил. Просматривая список выбранных технологий, я не мог понять, чем им "не понравилась" зависимость от СУБД:

- они выбрали язык программирования, привязанный к определенной операционной системе и поддерживаемый единственным производителем (можно было выбрать язык Java);
- они выбрали технологию создания компонентов, привязывающую к одной операционной системе и производителю (они могли выбрать технологию EJB или CORBA);
- они выбрали Web-сервер, работающий на единственной платформе того же производителя (почему не Apache?).

Все остальные технологии они выбрали так, что оказались привязанными к конкретной операционной системе — фактически свобода выбора оставалась только в отношении СУБД.

Независимо от того, что у них, видимо, были веские причины выбрать именно эти технологии, разработчики почему-то решили не использовать в полном объеме возможности критического компонента своей архитектуры и сделали это во имя "открытости". Мне кажется, что нужно сначала вдумчиво выбрать технологии, а затем максимально использовать предоставляемые ими возможности. За все эти технологии заплачены немалые деньги — не в ваших ли интересах максимально их использовать? Причем, создавалось впечатление, что они собирались воспользоваться преимуществами остальных технологий, так почему же для СУБД сделано исключение? На этот вопрос особенно сложно ответить, если учесть, что для эффективности приложения успешная работа с СУБД имеет первостепенное значение.

Можно рассмотреть это с точки зрения "открытости". Все данные помешаются в базу данных. СУБД, поддерживающая эту базу данных, — очень открытое средство. Она обеспечивает доступ к данным через SQL, с помощью компонетов EJB, по протоколам **HTTP,** FTP, SMB и с помощью множества других протоколов и механизмов доступа. Пока все отлично: что может быть более открытым?

Затем **вне** базы данных добавляются алгоритмы и, что важнее, механизмы **защиты.** Например, в компоненты, обеспечивающие доступ к данным, или в код на Visual Basic, работающий на сервере Microsoft Transaction Server (MTS). В результате с открытостью базы данных покончено — она уже "закрыта". Пользователи теперь не могут использовать эти данные с помощью существующих технологий — они **должны** использовать предложенные методы доступа (или обращаться к данным в обход защиты). Сегодня это не кажется проблемой, но помните: то, что сегодня является "самой современной" технологией, например компоненты **EJB,** вчера было идеей, а завтра будет устаревшей, неэффективной технологией. Что осталось неизменным за последние 20 с лишним лет в мире реляционного программирования (да, собственно, и объектно-оритентированного) — это базы данных. Средства работы для пользователей меняются практически ежегодно, и по мере этого все приложения, самостоятельно, а не с помощью СУБД, реализующие защиту, становятся препятствиями на пути дальнейшего прогресса.

СУБД Oracle предлагает возможность *тщательного контроля доступа* (Fine Grained Access Control, FGAC, — ему посвящена глава 21). Если коротко, эта технология позво-

ляет разработчику встраивать в базу данных процедуры, которые изменяют поступающие в базу данных запросы. Это изменение запросов используется для ограничения количества строк, которые клиент может получать или изменять. Процедура может определять, кто выполняет запрос, когда этот запрос выполняется, с какого терминала и т. д., и ограничивать соответствующим образом доступ к данным. С помощью FGAC можно организовать такую защиту, когда:

- запросы, выполняемые в нерабочее время определенным классом пользователей, не возвращают никаких записей;
- данные могут читаться с терминала в охраняемом офисе, но на терминал "удаленного" клиента конфиденциальная информация не выдается.

Эта возможность позволяет организовать контроль доступа в СУБД, **непосредственно выдающей данные.** Теперь уже неважно, получает ли пользователь данные через компоненты, страницы JSP, из приложения на VB с помощью ODBC или через SQL*Plus, — будут применяться одинаковые правила защиты. Вы готовы воспринять любую новую технологию.

Теперь я спрошу: какая технология более "открытая"? Та, что позволяет обращаться к данным только из кода VB и управляющих элементов ActiveX (замените язык VB языком Java, а компоненты ActiveX — компонентами EJB, если хотите; я говорю не о конретной технологии, а о подходе)? Или та, что обеспечивает доступ из любой среды, способной взаимодействовать с СУБД, по столь отличающимся протоколам, как SSL, HTTP и Net8, или с помощью функциональных интерфейсов ODBC, JDBC, OCI и т. д.? Покажите мне средство создания отчетов, способное выполнять запросы к коду на VB. Я же назову десятки таких средств, выполняющих SQL-запросы.

Решение идти на жертвы ради независимости от СУБД и полной "открытости" волен принять каждый, и многие так и поступают, но я считаю такое решение ошибочным. Независимо от СУБД, ее функциональные возможности необходимо использовать в полной мере. Именно это, как правило, и делается на этапе настройки производительности (этим приходится заниматься сразу же после внедрения). Удивительно, как быстро отказываются от требования независимости, если использование специфических возможностей СУБД позволяет ускорить работу приложения в пять раз.

Как ускорить работу?

Вынесенный в название раздела вопрос мне задают постоянно. Все ищут, где бы сделать установку **fast = true,** предполагая, что настройка производительности базы данных выполняется на уровне СУБД. Мой опыт показывает, что более 80 процентов (часто — намного больше, до 100 процентов) всего повышения производительности достигается на уровне приложения, а не базы данных. Нельзя заниматься настройкой СУБД, пока не настроено приложение, использующее данные.

Со временем появился ряд установок, включая которые на уровне СУБД, можно снизить влияние грубых ошибок программирования. Например, в Oracle 8.1.6 добавлен новый параметр — **CURSOR_SHARING=FORCE.** Он позволяет включить автоматическое использование связываемых переменных. В результате запрос SELECT * **FROM**

EMP WHERE EMPNO = 1234 автоматически переписывается в виде **SELECT * FROM EMP WHERE EMPNO = :x.** Это **может** существенно сократить количество жестких разборов и уменьшить ожидание защелок в библиотечном кэше, которые описаны в главе об архитектуре, но (всегда есть но) может также иметь ряд побочных эффектов. Можно нарваться на проблему (или ошибку) при использовании этой возможности, как, например, в первоначальной версии:

```
ops$tkyte@ORA8I.WORLD> alter session set cursor_sharing=force;
Session altered.

ops$tkyte@ORA8I.WORLD> select * from dual where dummy='X' and 1=0;
select * from dual where dummy='X' and 1=0
                                        *
ERROR at line 1:
ORA-00933: SQL command not properly ended

ops$tkyte@ORA8I.WORLD> alter session set cursor_sharing=exact;
Session altered.

ops$tkyte@ORA8I.WORLD> select * from dual where dummy='X' and 1=0;
no rows selected
```

Принятый способ переписывания запроса дает некорректный результат в версии 8.1.6 (из-за отсутствия пробела между X и ключевым словом **AND).** В итоге запрос приобретает вид:

```
select * from dual where dummy=:SYS_B_0 and :SYS_B_1=:SYS_B_2;
```

Ключевое слово **AND** стало частью имени связываемой переменной **:SYS_B_0. B** версии 8.1.7, однако, этот запрос переписывается так:

```
select * from dual where dummy=:"SYS_B_0" and :"SYS_B_1"=:"SYS_B_2";
```

Теперь **на** уровне **синтаксиса** все работает, но переписывание может отрицательно сказаться на производительности приложения. Например, обратите внимание, что в рассмотренном ранее коде условие **1=0** (всегда ложное) переписано как **:"SYS_B_1" = :"SYS_B_2".** Теперь на этапе анализа у оптимизатора нет полной информации чтобы определить, вернет ли этот запрос ноль строк (еще до его выполнения). Я понимаю, что запросов с конструкциями типа **1=0** у вас немного, но подозреваю, что в некоторых запросах литералы используются **умышленно. B** таблице может быть столбец с весьма неравномерным распределением значений (например, 90 процентов значений в столбце — больше 100, а 10 процентов — меньше 100). Причем лишь 1 процент значений меньше 50. Хотелось бы, чтобы при выполнении запроса:

```
select * from t where x < 50;
```

индекс использовался, а при выполнении запроса:

```
select * from t where x > 100;
```

не использовался. Если установить параметр **CURSOR_SHARING=FORCE,** оптимизатор не сможет учесть значения 50 или 100, поэтому будет выбирать план для общего

случая, когда индекс скорее всего не будет использоваться (даже если 99,9 процентов запросов будут содержать конструкцию WHERE x < 50).

Кроме того, я обнаружил, что, хотя установка CURSOR_SHARING = FORCE обеспечивает намного большую скорость работы, чем повторный анализ и оптимизация множества одинаковых запросов как уникальных, это все равно медленнее, чем выполнение запросов, где связываемые переменные используются изначально. Это происходит не из-за неэффективности механизма совместного использования кода курсора, а из-за неэффективности самой программы. В главе 10 мы рассмотрим, как разбор операторов SQL может влиять на производительность в целом. Во многих случаях приложение, не использующее связываемые переменные, также не обеспечивает эффективного анализа и повторного использования курсоров. Поскольку в приложении предполагается уникальность каждого запроса (так как для каждого из них создается уникальный оператор), то и курсор в нем не будет использоваться более одного раза. Факт в том, что если программист использует связываемые переменные, то он зачастую также разбирает запрос один раз и затем использует многократно. Именно затраты ресурсов на повторный разбор приводят к наблюдаемому снижению производительности.

Итак, важно помнить, что просто добавление параметра инициализации CURSOR_SHARING = FORCE не всегда позволяет решить проблемы. Могут даже возникнуть новые. Во многих случаях параметр CURSOR_SHARING — действительно полезное средство, но это не панацея. Для хорошо продуманного приложения он не нужен. В долгосрочной перспективе обоснованное использование связываемых переменных (и при необходимости — констант) — наиболее правильно.

Даже если есть соответствующие параметры, которые можно установить на уровне базы данных, а их пока немного, проблемы одновременного доступа и неэффективных запросов (неудачно сформулированных или вызванных неудачной организацией данных) нельзя решить только установкой параметров сервера. Для решения этих проблем необходимо переписать приложение (а зачастую и изменить его архитектуру). Перенос файлов данных с одного диска на другой, изменение количества блоков, читаемых подряд одной операцией ввода, и другие настройки "на уровне базы данных" часто мало влияют на общую производительность приложения. Они никак не дадут ускорения в 2, 3, ... N раз, необходимого для достижения приемлемой скорости работы приложения. Как часто требуется ускорить работу приложения на 10 процентов? Если надо ускорить работу на 10 процентов, обычно никто вообще не поднимает вопрос об этом. Пользователи начинают жаловаться, когда, по их мнению, скорость надо увеличить раз в пять. Однако повторяю: вы не увеличите скорость работы в пять раз за счет переноса файлов данных на другие диски. Это можно сделать только путем изменения приложения, например, сократив объем ввода/вывода.

О производительности необходимо думать уже на уровне проекта, а затем непрерывно проверять в процессе разработки. Это нельзя откладывать на потом. Я удивляюсь, сталкиваясь со случаями, когда разработчики передают приложение заказчику, устанавливают и только после этого начинают настраивать. Я видел приложения, которые поставлялись клиентам только с первичными ключами, вообще без дополнительных индексов. Запросы никто не настраивал и вообще не тестировал их производительность. С приложением никогда не работало более десятка пользователей. Настройка считается частью

процесса установки и внедрения программного продукта. Для меня такой подход неприемлем. Пользователи должны получать быстро работающую, хорошо настроенную систему. Проблем с продуктом у них будет достаточно и без производительности. Пользователи готовы к тому, что в приложении будут ошибки, но не заставляйте их бесконечно ждать появления сообщений об этих ошибках на экране.

Взаимоотношения АБД и разработчиков

На обложке книги сказано, как важно для АБД представлять, чего пытаются добиться разработчики, а для разработчиков — знать стратегию, используемую АБД для управления данными. Точно известно, что в основе большинства успешно работающих информационных систем лежит плодотворное взаимодействие между АБД и разработчиками приложений. В этом разделе я хочу представить точку зрения разработчика на разделение труда между разработчиком и АБД (исходя из предположения, что при любой важной разработке необходима поддержка группы АБД).

Разработчик не обязан знать, как устанавливать и конфигурировать программное обеспечение. Этим должен заниматься АБД и, возможно, системный администратор. Настройка Net8, запуск программы прослушивания, конфигурирование режима MTS, организация пула подключений, установка СУБД, создание базы данных и т.д. возлагаются на АБД и системного администратора.

Не обязан разработчик также уметь настраивать операционную систему. Лично я обычно предлагаю сделать это системным администраторам. Разработчик приложений баз данных должен быть квалифицированным пользователем соответствующей операционной системы, но нельзя требовать от него знания тонкостей ее настройки.

Пожалуй, одной из основных забот АБД является резервное копирование и восстановление базы данных, и я считаю это обязанностью исключительно АБД. А вот знать принцип работ и использования сегментов отката и журналов повторного выполнения разработчик должен. Знать, как выполнить восстановление табличного пространства по состоянию на определенный момент времени разработчику необязательно. Знание того, что это в принципе возможно, может пригодиться, но делать это самостоятельно вам не придется.

Настройка на уровне экземпляра базы данных, определение оптимального значения параметра SORT_AREA_SIZE — этим обычно занимается АБД. Бывают ситуации, когда разработчику необходимо изменить ряд параметров сеанса, но за параметры уровня базы данных отвечает АБД. Обычно база данных поддерживает приложения нескольких разработчиков, поэтому только АБД, занимающийся поддержкой всех приложений, может принять правильное решение.

Выделение пространства на диске и управление файлами данных — обязанность АБД. Разработчики должны оговорить необходимый объем пространства (сколько им предположительно потребуется), но остальное должны делать АБД и системный администратор.

Итак, разработчики могут не знать, как запустить СУБД, но должны уметь работать в ней. Разработчик и АБД совместно решают разные части одной головоломки. АБД связывается с разработчиком, заметив, что запросы потребляют слишком много ресурсов, а разработчик обычно обращается к АБД когда не знает, как ускорить работу системы

(вот когда занимаются настройкой экземпляра — когда приложение полностью настроено).

Конечно, в зависимости от среды разработки возможны варианты, но мне нравится делить обязанности. Хороший разработчик обычно — очень плохой АБД, и наоборот. У них разные навыки и опыт, а также, по моим наблюдениям, разное устройство ума и личностные характеристики.

Резюме

Мы в общих чертах рассмотрели, почему необходимо знать используемую СУБД. Приведенные примеры — не уникальны, подобное происходит на практике каждый день. Давайте кратко повторим ключевые моменты. Если вы разрабатываете ПО для СУБД Oracle:

- Вы должны понимать архитектуру Oracle. Не требуется знать ее настолько, чтобы переписать сервер, но достаточно хорошо, чтобы понимать последствия использования тех или иных возможностей.
- Необходимо понимать, как выполняется блокирование и управление одновременным доступом, и учитывать, что в каждой СУБД это реализуется по-разному. Без этого понимания СУБД будет давать "неверные" ответы и у вас будут большие проблемы с конфликтами доступа и, как следствие, — низкая производительность.
- Не воспринимайте СУБД как черный ящик, устройство которого понимать не обязательно. СУБД — самая важная часть большинства приложений. Ее игнорирование приводит к фатальным последствиям.
- Не изобретайте велосипед. Я встречал разработчиков, попавших в трудное положение не только технически, но и на личном уровне из-за незнания возможностей СУБД Oracle. Это происходило, когда оказывалось, что реализуемые ими в течение нескольких месяцев функции на самом деле давно встроены в СУБД.
- Решайте проблемы как можно проще, максимально используя встроенные возможности СУБД Oracle. Вы немало заплатили за это.

Программные проекты начинаются и заканчиваются, языки и среды программирования появляются и исчезают. От нас, разработчиков, ждут создания работающих систем в течение недель, может быть, месяцев, а затем мы переходим к следующей задаче. Если мы будем каждый раз изобретать велосипед, то никогда не перейдем к сути разработки. Никто ведь не создает класс, реализующий хеш-таблицу в Java, — он входит в набор стандартных компонентов. Вот и используйте имеющиеся функциональные возможности СУБД. Первый шаг к этому — узнать их. Читайте дальше.

2

Архитектура

Oracle проектировалась как максимально переносимая СУБД, — она доступна на всех распространенных платформах. Поэтому физическая архитектура Oracle различна в разных операционных системах. Например, в ОС UNIX СУБД Oracle реализована в виде нескольких отдельных процессов операционной системы — практически каждая существенная функция реализована отдельным процессом. Для UNIX такая реализация подходит, поскольку основой многозадачности в ней является процесс. Для Windows, однако, подобная реализация не подходит и работала бы не слишком хорошо (система получилась бы медленной и плохо масштабируемой). На этой платформе СУБД Oracle реализована как один многопоточный процесс, т.е. с использованием подходящих для этой платформы механизмов реализации. На больших ЭВМ IBM, работающих под управлением OS/390 и zOS, СУБД Oracle использует несколько адресных пространств OS/390, совместно образующих экземпляр Oracle. Для одного экземпляра базы данных можно сконфигурировать до 255 адресных пространств. Более того, СУБД Oracle взаимодействует с диспетчером загрузки OS/390 WorkLoad Manager (WLM) для установки приоритетности выполнения определенных компонентов Oracle по отношению друг к другу и к другим задачам, работающим в системе OS/390. В ОС Netware тоже используется многопоточная модель. Хотя физические средства реализации СУБД Oracle на разных платформах могут отличаться, архитектура системы — достаточно общая, чтобы можно было понять, как СУБД Oracle работает на всех платформах.

В этой главе мы рассмотрим три основных компонента архитектуры Oracle.

- **Файлы.** Будут рассмотрены пять видов файлов, образующих базу данных и поддерживающих экземпляр. Это файлы *параметров, сообщений, данных, временных данных* и *журналов повторного выполнения*.
- **Структуры памяти,** в частности системная глобальная область (System Global Area — SGA). Мы рассмотрим взаимодействие SGA, PGA и UGA. Будут также рассмотрены входящие в SGA Java-пул, разделяемый пул и большой пул.
- **Физические процессы или потоки.** Будут описаны три типа процессов, образующих экземпляр: *серверные* процессы, *фоновые* процессы и *подчиненные* процессы.

Сервер

Трудно решить, с какого компонента сервера начать описание. Процессы используют область SGA, поэтому рассматривать SGA до процессов не имеет смысла. С другой стороны, при описании процессов и их функционирования придется ссылаться на компоненты SGA. Они тесно взаимосвязаны. С файлами работают процессы, и их нет смысла описывать, пока не объяснено, что делают процессы. Ниже определены некоторые термины и сделан общий обзор сервера Oracle, после чего подробно рассматриваются отдельные компоненты.

Два термина в контексте Oracle вызывают большую путаницу. Речь идет о терминах "база данных" и "экземпляр". В соответствии с принятой в Oracle терминологией, эти понятия определяются так:

- *база данных* — набор физических файлов операционной системы;
- *экземпляр* — набор процессов Oracle и область SGA.

Эти два термина иногда взаимозаменяемы, но представляют принципиально разные концепции. Взаимосвязь между ними такова, что база данных может быть смонтирована и открыта в нескольких экземплярах. Экземпляр может смонтировать и открыть только одну базу данных в каждый момент времени. Не обязательно отрывать и монтировать одну и ту же базу данных при каждом запуске экземпляра.

Стало еще непонятнее? Вот ряд примеров, которые помогут прояснить ситуацию. Экземпляр — это набор процессов операционной системы и используемая ими память. Все эти процессы могут работать с базой данных, которая представляет собой просто набор файлов (файлов данных, временных файлов, файлов журнала повторного выполнения, управляющих файлов). В каждый момент времени с экземпляром связан только один набор файлов. В большинстве случаев обратное утверждение тоже верно; с базой данных работает только один экземпляр. В случае же использования параллельного сервера Oracle (Oracle Parallel Server — OPS), опции Oracle, позволяющей серверу функционировать на нескольких компьютерах в кластерной среде, одна и та же база данных может быть одновременно смонтирована и открыта несколькими экземплярами. Это делает возможным доступ к базе данных одновременно с нескольких компьютеров. Oracle Parallel Server позволяет создавать системы с высокой доступностью данных и, при условии правильной реализации, очень масштабируемые. Рассмотрение опции OPS здесь

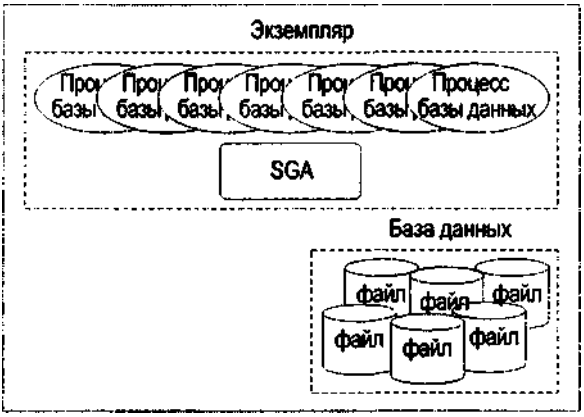
не предусмотрено, поскольку для описания особенностей ее реализации потребовалась бы отдельная книга.

Итак, в большинстве случаев между базой данных и экземпляром имеется отношение один к одному. Это, вероятно, и является причиной путаницы при использовании этих терминов. По опыту большинства пользователей, база данных — это экземпляр, а экземпляр — это база данных.

Во многих тестовых средах это, однако, не так. На моем диске, например, может быть пять отдельных баз данных. На тестовой машине СУБД Oracle установлена в одном экземпляре. В каждый момент времени работает только один экземпляр, но обращаться он может к разным базам данных, в зависимости от задач, которые я решаю. Создав несколько конфигурационных файлов, я могу монтировать и открывать любую из этих баз данных. В данном случае у меня один "экземпляр", но несколько баз данных, лишь одна из которых доступна в каждый момент времени.

Итак, теперь под термином "экземпляр" мы будем понимать процессы и память сервера Oracle. Термин "база данных" означает физические файлы, в которых находятся данные. База данных может быть доступна многим экземплярам, но экземпляр в каждый момент времени обеспечивает доступ только к одной базе данных.

Теперь можно приступать к рассмотрению абстрактной схемы СУБД Oracle.



Это упрошенный вид СУБД Oracle. Она включает большую область памяти — SGA, — содержащая внутренние структуры данных, доступ к которым необходим всем процессам для кэширования данных с диска, кэширования данных повторного выполнения перед записью на диск, хранения планов выполнения разобранных операторов SQL и т.д. Имеется также набор процессов, подключенных к этой области SGA, причем механизм подключения каждой операционной системы другой. В среде UNIX процесс физически подключаются к большому сегменту разделяемой памяти — выделенному ОС фрагменту памяти, к которому может одновременно обращаться несколько процессов. В ОС Windows для выделения памяти процессы используют библиотечную функцию **malloc()** языка С, поскольку они сами являются потоками одного большого процесса. В СУБД Oracle также имеется набор файлов, читаемых и записываемых процессами/потоками базы данных (причем читать и записывать эти файлы имеют право только про-

цессы Oracle). В этих файлах хранятся данные таблиц, индексов, временное пространство, журналы повторного выполнения и т.д.

Если запустить СУБД Oracle в UNIX-системе и выполнить команду ps (для просмотра состояния процессов), можно увидеть количество работающих процессов и их имена. Например:

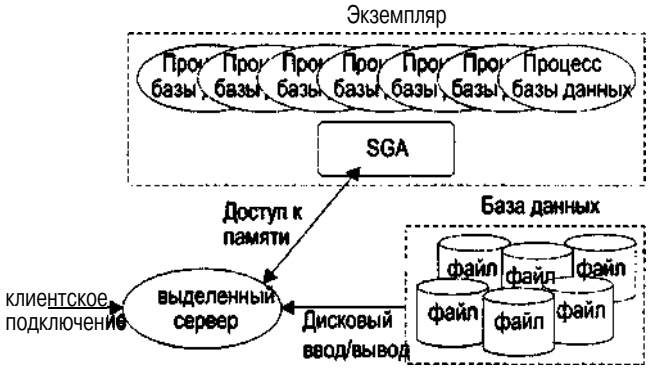
```
$ /bin/pe -aef | grep ora816
  ora816 20827     1  0   Feb 09 ?       0:00 ora_d000_ora816dev
  ora816 20821     1  0   Feb 09 ?       0:06 ora_smon_ora816dev
  ora816 20817     1  0   Feb 09 ?       0:57 ora_lgwr_ora816dev
  ora816 20813     1  0   Feb 09 ?       0:00 ora_pmon_ora816dev
  ora816 20819     1  0   Feb 09 ?       0:45 ora_ckpt_ora816dev
  ora816 20815     1  0   Feb 09 ?       0:27 ora_dbw0_ora816dev
  ora816 20825     1  0   Feb 09 ?       0:00 ora_s000_ora816dev
  ora816 20823     1  0   Feb 09 ?       0:00 ora_reco_ora816dev
```

Я еще опишу назначение каждого из этих процессов, но часто их в совокупности называют просто *фоновыми процессами Oracle.* Это — постоянно работающие процессы, образующие экземпляр; они появляются при запуске СУБД и работают до тех пор, пока она не будет остановлена. Интересно отметить, что все это — процессы, а не программы. СУБД Oracle реализуется одной программой в UNIX, но программа эта многолика. Программа, которая запускалась для реализации процесса **ora_lgwr_ora816dev,** была использована и для запуска процесса **ora_ckpt_ora816dev.** Есть только один двоичный файл с именем **oracle.** Просто он выполняется несколько раз с разными именами. В ОС Windows с помощью программы tlist, входящей в Windows resource toolkit, можно обнаружить только один процесс — Oracle.exe. В случае NT тоже есть всего одна двоичная программа. Этот процесс создает несколько потоков, представляющих фоновые процессы Oracle. С помощью утилиты tlist (или любого из множества подобных средств) можно увидеть эти потоки:

```
C:\Documents and Settings\Thomas Kyte\Desktop>tlist 1072
1072 ORACLE.EXE
    CTO:      C:\oracle\DATABASE\
    CmdLine:  c:\oracle\bin\ORACLE.EXE TKYTE816
    VirtualSite:    144780 KB    PeakVirtualSize:    154616 KB
    WorkingSetSize:  69424 KB    PeakNorkingSetSize:  71208 KB
    NumberOfThreads: 11
        0 Win32startAddr:0x00000000 LastErr:0x00000000 State:Initialized
        5 Win32StartAddr:0x00000000 LastErr:0x00000000 State:Initialized
        5 Win32StartAddr:0x00000000 LastErr:0x00000000 State:Initialized
        5 Hin32StartAddr:0x00000000 LastErr:0x00000000 State:Initialized
        5 Win32stertAddr:0x00000000 LastErr:0x00000000 State:Initialized
        5 Win32StartAddx:0x00000000 LastErr:0x00000000 State:Initialized
        5 Win32StartAddr:0x00000000 LastErr:0x00000000 State:Initialized
        5 Win32StartAddr:0x00000000 LastErr:0x00000000 State:Initialized
        5 Win32StartAddr:0x00000000 LastErr:0x00000000 State:Initialized
        5 Win32StartAddr:0x00000000 LastErr:0x00000000 State:Initialized
        5 Win32StartAddr:0x00000000 LastErr:0x00000000 State:Initialized
          0.0.0.0  shp0x00400000    ORACLE.EXE
        5.0.2163.1  shp0x77f80000    ntdll.dll
          0.0.0.0  shp0x60400000    oraclient8.dll
```

```
0.0.0.0 shp0x60600000    oracore8.dll
0.0.0.0 shp0x60800000    oranls8.dll
```

В данном случае имеется 11 потоков, выполняющихся в рамках одного процесса Oracle. Если подключиться к базе данных, количество потоков увеличится до 12. В ОС UNIX к существующим процессам **oracle** просто добавился бы еще один. Теперь можно представить следующую схему. Предыдущая схема представляла концептуальный вид СУБД Oracle сразу после запуска. Теперь, если подключиться к СУБД Oracle в наиболее типичной конфигурации, схема будет выглядеть примерно так:

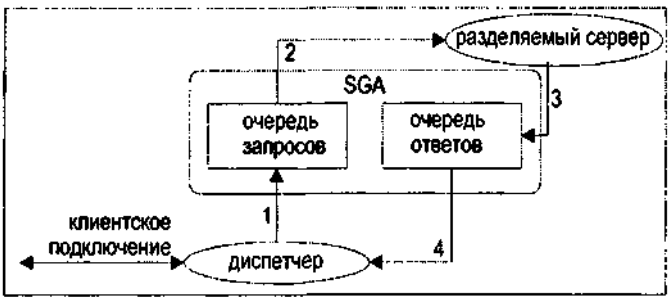


Обычно СУБД Oracle при подключении пользователя создает новый процесс. Это принято называть конфигурацией *выделенного сервера,* поскольку на все время сеанса ему выделяется отдельный серверный процесс. Сеансы и выделенные серверы находятся в отношении один к одному. Клиентский процесс (любая программа, пытающаяся подключиться к СУБД) будет непосредственно взаимодействовать с соответствующим выделенным сервером по сети, например, через сокет **TCP/IP.** Именно этот сервер будет получать и выполнять SQL-операторы. Он будет читать файлы данных, а также искать необходимые данные в кэше. Он будет выполнять операторы **UPDATE** и PL/SQL-код. Единственное его назначение — отвечать на получаемые SQL-запросы.

СУБД Oracle также может работать в режиме многопотокового сервера (multi-threaded server — MTS), в котором при подключении не создается дополнительный поток или процесс **UNIX. В** режиме MTS СУБД Oracle использует пул "разделяемых серверов" для поддержки большого количества пользователей. Разделяемые серверы — это просто механизм организации пула подключений. Вместо запуска 10000 выделенных серверов (это действительно много, если речь идет о процессах или потоках) для 10000 сеансов режим **MTS** позволяет обслуживать их с помощью гораздо меньшего количества разделяемых серверов, которые (как следует из названия) будут совместно использоваться всеми сеансами. Это позволяет СУБД Oracle поддерживать намного больше сеансов, чем в режиме выделенного сервера. Машина, на которой работает сервер, может не справиться с поддержкой 10000 процессов, но управление 100 или 1000 процессами для нее вполне реально. В режиме MTS разделяемые серверные процессы обычно запускаются сразу при

старте СУБД и отображаются в списке, выдаваемом командой ps (в представленных выше результатах выполнения команды ps процесс **ora_s000_ora816dev** представляет собой разделяемый серверный процесс).

Принципиальное отличие режима MTS от режима выделенного сервера состоит в том, что клиентский процесс, подключившийся к СУБД, никогда не взаимодействует непосредственно с разделяемым сервером, как это происходит в случае выделенного сервера. Он не может взаимодействовать с разделяемым сервером, так как соответствующий процесс используется совместно. Чтобы обеспечить совместное использование этих процессов, необходим другой механизм взаимодействия. Для этого **в СУБД Oracle** используется процесс (или набор процессов), которые называют *диспетчерами.* Клиентский процесс взаимодействует по сети с процессом-диспетчером. Процесс-диспетчер помещает запрос клиента в очередь запросов в SGA (это одно из многих назначений области SGA). Первый же свободный разделяемый сервер выберет и обработает этот запрос (например, запрос может иметь вид **UPDATE T SET X = X+5 WHERE Y = 2).** По завершении выполнения команды разделяемый сервер поместит ответ в очередь ответов. Процесс-диспетчер следит за очередью и немедленно передает полученный результат клиенту. Концептуально поток информации в режиме MTS выглядит следующим образом:



Клиентское подключение посылает запрос диспетчеру. Диспетчер поместит этот запрос в очередь запросов в области SGA (1). Первый свободный разделяемый сервер выберет этот запрос (2) из очереди и обработает его. Когда разделяемый сервер закончит выполнение, ответ (коды возврата, данные и т.д.) помещается в очередь ответов (3), после чего выбирается диспетчером (4) и возвращается клиенту.

С точки зрения разработчика нет никакой разницы между подключением к серверу в режиме MTS и подключением к выделенному серверу. Теперь, когда стало понятно, как происходит подключение к выделенному и разделяемому серверу, возникают вопросы: а как вообще подключиться; как запускается выделенный сервер и как связываться с процессом-диспетчером? Ответы зависят от платформы, но в принципе все происходит так, как описано ниже.

Мы рассмотрим наиболее общий случай: запрос на подключение по сети с использованием протоколов TCP/IP. В этом случае клиент находится на одной машине, а сервер — на другой, причем эти машины связаны сетью на базе семейства протоколов **TCP/IP.** Все начинается с клиента. Он посылает запрос клиентскому ПО Oracle на подключение к базе данных. Например, выполняется команда:

```
C:\> sqlplus  scott/tiger@ora816.us.oracle.com
```

Здесь клиентом является утилита SQL*Plus. **scott/tiger** — имя пользователя и пароль, а **ora816.us.oracle.com** — имя службы TNS. TNS — сокращение от Transparent Network Substrate (прозрачная сетевая среда), которое обозначает "базовое" программное обеспечение, встроенное в клиент Oracle и обеспечивающее удаленное подключение (двухточечное взаимодействие клиента и сервера). Строка подключения TNS указывает программному обеспечению Oracle, как подключаться к удаленной базе данных. В общем случае клиентское программное обеспечение обращается к файлу TNSNAMES.ORA. Это обычный текстовый файл конфигурации, обычно находящийся в каталоге **[ORACLE_HOME]\network\admin** и содержащий записи вида:

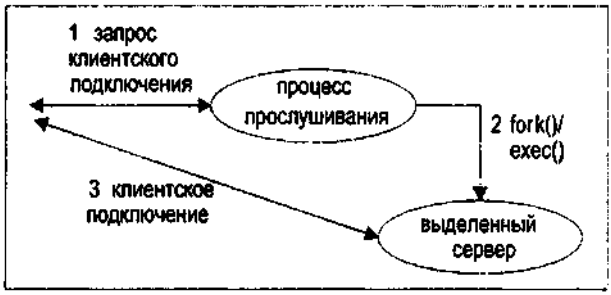
```
ORA816.US.ORACXE.COM =
   (DESCRIPTION =
      (ADDRESS_LIST =
         (ADDRESS = (PROTOCOL = TCP) (HOST = aria.us.oracle.com)(PORT = 1521))
      )
      (CONNECT_DATA =
         (ORACLE_SID = ora816)
      )
   )
```

Именно эти параметры конфигурации позволяют клиентскому ПО Oracle преобразовать строку **ora816.us.oracle.com** в необходимые для подключения данные: имя хоста; порт на этом хосте, прослушиваемый процессом, который принимает подключения; идентификатор SID (Site IDentifier) базы данных на хосте, к которой необходимо подключиться, и т.д. Эта строка, **ora816.us.oracle.com,** может преобразовываться в необходимые данные и по-другому. Например, она может преобразовываться с помощью службы Oracle Names — распределенного сервера имен для СУБД, аналогичного по назначению службе DNS, используемой для преобразования имен хостов в IP-адреса. Однако в большинстве небольших и средних серверов, где количество копий конфигурационных файлов невелико, чаще всего используется именно файл TNSNAMES.ORA.

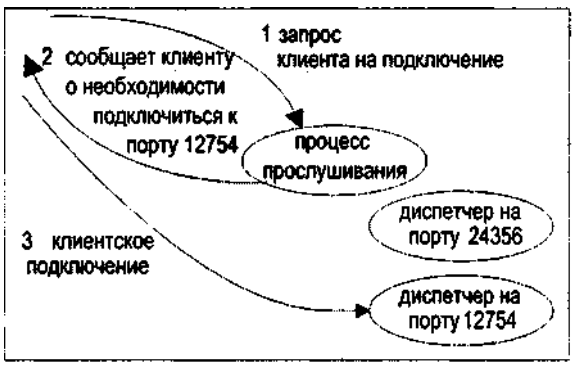
Теперь, когда клиентскому ПО известно, куда подключаться, оно открывает соединение через сокет TCP/IP к порту 1521 машины **aria.us.oracle.com.** Если администратор базы данных соответствующего сервера настроил службу Net8 и запустил процесс прослушивания, это подключение может быть принято. В сетевой среде на сервере работает процесс TNS **Listener.** Это процесс прослушивания, обеспечивающий физическое подключение к базе данных. Получив запрос на подключение, он проверяет его, используя собственные файлы конфигурации, и либо отвечает отказом (например, не существует запрашиваемой базы данных или IP-адрес подключающегося содержится в списке тех, кому не разрешено подключение к хосту), либо обеспечивает подключение клиента.

При подключении к выделенному серверу процесс прослушивания автоматически запустит выделенный сервер. В ОС UNIX это делается с помощью системных вызовов fork() и exec() (единственный способ создать новый процесс после инициализации ОС UNIX — использовать системный вызов **fork()).** Теперь мы физически подключены к базе данных. В Windows процесс прослушивания требует от серверного процесса создания нового потока для подключения. После создания этого потока клиент "перенаправ-

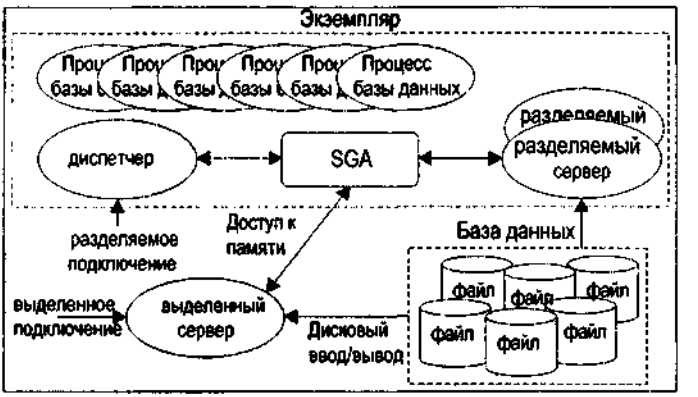
ляется" на него, и тем самым обеспечивается физическое подключение. В случае ОС UNIX это можно представить следующей схемой:



В режиме MTS процесс прослушивания работает иначе. Ему известно, какие процессы-диспетчеры работают в составе экземпляра. При получении запроса на подключение процесс прослушивания выбирает процесс-диспетчер из пула доступных диспетчеров. Затем он посылает клиенту информацию, позволяющую подключиться к процессу-диспетчеру. Это необходимо, поскольку процесс прослушивания работает на известном порту соответствующего хоста, а вот диспетчеры будут принимать подключения через произвольно выделенные порты. Процессу прослушивания известны эти выделенные порты, поэтому он автоматически выбирает свободный диспетчер. Затем клиент отключается от процесса прослушивания и подключается непосредственно к диспетчеру. В результате устанавливается физическое соединение с СУБД. Графически это можно представить так:



Итак, обзор архитектуры Oracle закончен. Мы описали, что такое экземпляр Oracle, что такое база данных и как можно подключиться к базе данных через выделенный и разделяемый сервер. На следующей схеме показано взаимодействие с сервером Oracle клиента, подключенного к разделяемому серверу, и клиента, работающего с выделенным серверным процессом. Один экземпляр Oracle может поддерживать оба типа подключений одновременно:



Теперь подробно рассмотрим процессы, образующие сервер, их назначение и взаимодействие друг с другом, а также содержимое области SGA и назначение ее компонентов. Но начнем мы с описания различных типов файлов, которые сервер Oracle использует для управления данными.

Файлы

В состав базы данных и экземпляра входит шесть типов файлов. С экземпляром связаны *файлы параметров.* По этим файлам экземпляр при запуске определяет свои характеристики, например размер структур в памяти и местонахождение управляющих файлов.

Базу данных образуют следующие файлы.

- **Файлы данных.** Собственно данные (в этих файлах хранятся таблицы, индексы и все остальные сегменты).
- **Файлы журнала повторного выполнения.** Журналы транзакций.
- **Управляющие файлы.** Определяют местонахождение файлов данных и содержат другую необходимую информацию о состоянии базы данных.
- **Временные файлы.** Используются при сортировке больших объемов данных и для хранения временных объектов.
- **Файлы паролей.** Используются для аутентификации пользователей, выполняющих администрирование удаленно, по сети. Мы не будем их подробно рассматривать.

Наиболее важны первые два типа файлов, поскольку именно в них хранятся накопленные данные. В случае потери остальных файлов хранящиеся данные не пострадают. Если будут потеряны файлы журнала повторного выполнения, **некоторые** данные могут быть потеряны. Если же будут потеряны файлы данных и все их резервные копии, данные, **безусловно,** будут потеряны навсегда.

Теперь давайте детально рассмотрим все типы файлов и их содержимое.

Файлы параметров

С базой данных Oracle связано много файлов параметров: от файла **TNSNAMES.ORA** на клиентской рабочей станции (используемого для поиска сервера) и файла **LISTENER.ORA** на сервере (для запуска процесса прослушивания Net8) до файлов **SQLNET.ORA, PROTOCOL.ORA, NAMES.ORA, CMAN.ORA** и **LDAP.ORA.** Наиболее важным является файл параметров инициализации экземпляра, потому что без него не удастся запустить экземпляр. Остальные файлы тоже важны; они связаны с поддержкой сети и обеспечением подключения к базе данных. Однако их рассмотрение в этом разделе они не будут. Сведения об их конфигурировании и настройке можно найти в руководстве *Oracle Net8 Administrators Guide*. Обычно разработчик не настраивает эти файлы — они создаются администратором.

Файл параметров инициализации экземпляра обычно называют файлом **init** или файлом **init.ora.** Это название происходит от стандартного имени этого файла, — **init<ORACLE_SID>.ora.** Например, экземпляр со значением **SID,** равным **tkyte816,** обычно имеет файл инициализации **inittkyte816.ora.** Без файла параметров инициализации нельзя запустить экземпляр Oracle. Поэтому файл этот достаточно важен. Однако, поскольку это обычный текстовый файл, который можно создать в любом текстовом редакторе, сохранять его ценой собственной жизни не стоит.

Для тех, кому незнаком термин **SID** или параметр **ORACLE_SID,** представлю полное определение. **SID** — это идентификатор экземпляра (сайта). В ОС UNIX он хешируется совместно со значением **ORACLE_HOME** (задающим каталог, в котором установлено ПО Oracle) для создания уникального ключа при подсоединении области SGA. Если значение **ORACLE_SID** или **ORACLE_HOME** задано неправильно, выдается сообщение об ошибке **ORACLE NOT AVAILABLE,** поскольку невозможно подключиться к сегменту разделяемой памяти, определяемому этим "магическим" ключом. В ОС Windows разделяемая память используется не так, как в ОС UNIX, но параметр **SID** все равно важен. В одном и том же базовом каталоге **ORACLE_HOME** может быть несколько баз данных, так что необходимо иметь возможность уникально идентифицировать их и соответствующие конфигурационные файлы.

В Oracle файл **init.ora** имеет очень простую конструкцию. Он представляет собой набор пар имя параметра/значение. Файл **init.ora** может иметь такой вид:

```
db_name = "tkyte816"
db_block_size = 8192
control_files = ("C:\oradata\control01.ctl", "C:\oradata\control02.ctl")
```

Фактически это почти минимальный файл **init.ora,** с которым уже можно работать. В нем указан размер блока, стандартный для моей платформы (стандартный размер блока различен для разных платформ), так что я могу эту строку удалить. Файл параметров инициализации используется для получения имени базы данных и местонахождения управляющих файлов. Управляющие файлы содержат информацию о местонахождении всех остальных файлов, так что они нужны в процессе начальной загрузки при запуске экземпляра.

В файле параметров инициализации обычно содержится и много других параметров. Количество и имена параметров меняются с каждой новой версией. Например, в Oracle 8.1.5 был параметр **plsql_load_without_compile.** Его не было ни в одной из предыдущих версий и нет в последующих. В моих базах данных версий 8.1.5, 8.1.6 и 8.1.7 имеется, соответственно, 199, 201 и 203 различных параметра инициализации. Большинство параметров, например **db_block_size,** существует очень давно (они были во всех версиях), но со временем необходимость во многих параметрах отпадает, так как меняется реализация. Если захочется прочитать об этих параметрах и разобраться, что они позволяют установить, обратитесь к руководству *Oracle8i Reference.* В первой главе этого руководства представлены официально поддерживаемые параметры инициализации.

Обратите внимание на слова "официально поддерживаемые" в предыдущем абзаце. Не поддерживаются (и не описаны в руководстве) параметры, имена которых начинаются с символа подчеркивания. Вокруг этих параметров много спекуляций: поскольку они не поддерживаются официально, значит, имеют "магические" свойства. Многие полагают, что эти параметры хорошо известны "посвященным" сотрудникам корпорации Oracle и используются ими. По моему мнению, все как раз наоборот. Их мало кто вообще знает и редко использует. Большинство из неописанных параметров — лишние, представляют устаревшие возможности или обеспечивают обратную совместимость. Другие помогают при восстановлении данных, но не всей базы данных: они позволяют запустить экземпляр в определенных экстремальных ситуациях, но лишь для **извлечения** данных — базу данных затем придется пересоздавать. Я не вижу смысла использовать неописанные параметры файла **init.ora** в реальной базе данных, если только этого не требует служба технической поддержки. Многие из них имеют побочные эффекты, которые могут оказаться разрушительными. В базе данных, которую я использую для разработки, установлен только один неописанный параметр:

```
_TRACE_FILES_PUBLIC = TRUE
```

Это делает трассировочные файлы доступными всем, а не только членам группы **dba.** Я хочу, чтобы разработчики как можно чаще использовали установки **SQL_TRACE, TIMED_STATISTICS** и утилиту **TKPROF** (более того, я это требую), поэтому всем им необходимо читать трассировочные файлы. В производственной базе данных я неописанных параметров не использую.

Неописанные параметры должны использоваться только по указанию службы технической поддержки Oracle. При их использовании можно повредить базу данных, да и реализация меняется от версии к версии.

Теперь, когда известно, что представляют собой файлы параметров и где можно более подробно прочитать о параметрах, которые в них можно устанавливать, осталось узнать, где эти файлы искать на диске. Файлы параметров инициализации экземпляра принято именовать так:

```
init$ORACLE_SID.ora       (переменная среды Unix)
init%ORACLE_SID%.ora      (переменная среды Windows)
```

Как правило, они находятся в каталогах

```
$ORACLE_HOME/dbs          (в ОС Unix)
%ORACLE HOME%\DATABASE    (в ОС Windows)
```

Часто в файле параметров содержится всего одна строка примерно такого вида:

```
IFILE='C:\oracle\admin\tkyte816\pfile\init.ora'
```

Директива **IFILE** работает аналогично директиве препроцессора **#include** в языке C. Она вставляет в данном месте текущего файла содержимое указанного файла. В данном случае включается содержимое файла **init.ora** из нестандартного каталога.

Следует отметить, что файл параметров не обязательно должен находится в одном и том же стандартном месте. При запуске экземпляра можно использовать параметр **pfile = имя_файла.** Это особенно полезно при попытке проверить результаты установки других значений для параметров.

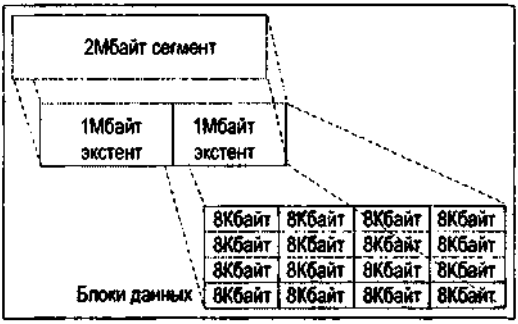
Файлы данных

Файлы данных вместе с файлами журнала повторного выполнения являются наиболее важными в базе данных. Именно в них хранятся все данные. В каждой базе данных есть хотя бы один файл данных, но обычно их намного больше. Только самые простые, "тестовые" базы данных имеют один файл данных. В любой реальной базе данных должно быть **минимум** два файла данных: один — для системных данных (табличное пространство SYSTEM), другой — для пользовательских (табличное пространство **USER).** В этом разделе мы рассмотрим организацию файлов данных в Oracle и способы хранения данных в этих файлах. Но прежде надо разобраться, что такое табличное пространство, сегмент, экстент и блок. Все это — единицы выделения пространства под объекты в базе данных Oracle.

Начнем с сегментов. *Сегменты* — это области на диске, выделяемые под объекты — таблицы, индексы, сегменты отката и т.д. При создании таблицы создается сегмент таблицы. При создании фрагментированной таблицы создается по сегменту для каждого фрагмента. При создании индекса создается сегмент индекса и т.д. Каждый объект, занимающий место на диске, хранится в одном сегменте. Есть сегменты отката, временные сегменты, сегменты кластеров, сегменты индексов и т.д.

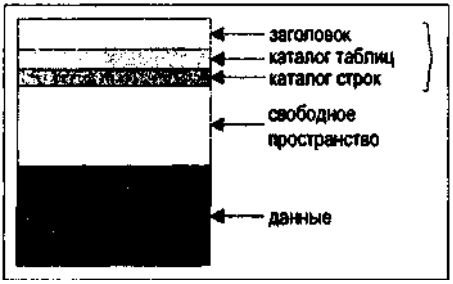
Сегменты, в свою очередь, состоят из одного или нескольких экстентов. *Экстент* — это непрерывный фрагмент пространства в файле. Каждый сегмент первоначально состоит хотя бы из одного экстента, причем для некоторых объектов требуется минимум два экстента (в качестве примера можно назвать сегменты отката). Чтобы объект мог вырасти за пределы исходного экстента, ему необходимо выделить следующий экстент. Этот экстент не обязательно должен выделяться рядом с первым; он может находиться достаточно далеко от первого, но в пределах экстента в файле пространство всегда непрерывно. Размер экстента варьируется от одного блока до 2 Гбайт.

Экстенты состоят из блоков. *Блок* — наименьшая единица выделения пространства в Oracle. В блоках и хранятся строки данных, индексов или промежуточные результаты сортировок. Именно блоками сервер Oracle обычно выполняет чтение и запись на диск. Блоки в Oracle бывают размером 2 Кбайт, 4 Кбайт или 8 Кбайт (хотя допустимы также блоки размером 16 Кбайт и 32 Кбайт). Отношения между сегментами, экстентами и блоками показаны на следующей схеме:



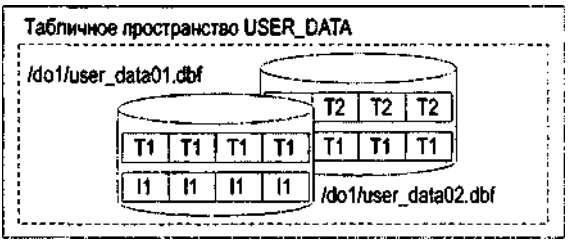
Сегмент состоит из одного или более экстентов, а экстент — это группа следующих подряд блоков.

Размер блока в базе данных с момента ее создания — величина постоянная, поэтому все блоки в базе данных одного размера. Формат блока представлен ниже.



Заголовок блока содержит информацию о типе блока (блок таблицы, блок индекса и т.д.), информацию о текущих и прежних транзакциях, затронувших блок, а также адрес (местонахождение) блока на диске. *Каталог таблиц* содержит информацию о таблицах, строки которых хранятся в этом блоке (в блоке могут храниться данные нескольких таблиц). *Каталог строк* содержит описание хранящихся в блоке строк. Это массив указателей на строки, хранящиеся в области данных блока. Вместе эти три части блока называются *служебным пространством блока*. Это пространство недоступно для данных и используется сервером Oracle для управления блоком. Остальные две части блока вполне понятны: в блоке имеется *занятое* пространство, в котором хранятся данные, и может быть *свободное* пространство.

Теперь, получив общее представление о сегментах, состоящих из экстентов, которые, в свою очередь, состоят из блоков, можно переходить к понятию *табличное пространство* и разбираться, где же в этой структуре место для файлов. Табличное пространство — это контейнер с сегментами. Каждый сегмент принадлежит к одному табличному пространству. В табличном пространстве может быть много сегментов. Все экстенты сегмента находятся в табличном пространстве, где создан сегмент. Сегменты никогда не переходят границ табличного пространства. С табличным пространством, в свою очередь, связан один или несколько файлов данных. Экстент любого сегмента табличного пространства целиком помещается в одном файле данных. Однако экстенты сегмента могут находиться в нескольких различных файлах данных. Графически это можно представить следующим образом:



Итак, здесь представлено табличное пространство **USER_DATA**. Оно состоит из двух файлов данных — **user_data01** и **user_data02. В** нем выделено три сегмента: T1, T2 и I1 (вероятно, две таблицы и индекс). В табличном пространстве выделены четыре экстента, причем каждый показан как непрерывный набор блоков базы данных. Сегмент T1 состоит из двух экстентов (по одному экстенту в каждом файле). Сегменты T2 и И состоят из одного экстента. Если для табличного пространства понадобится больше места, можно либо увеличить размер файлов данных, уже выделенных ему, либо добавить третий файл данных.

Табличные пространства в Oracle — это логические структуры хранения данных. Разработчики создают сегменты в табличных пространствах. Они никогда не переходят на уровень файлов — нельзя указать, что экстенты должны выделяться из определенного файла. Объекты создаются в табличных пространствах, а об остальном заботится сервер Oracle. Если в дальнейшем администратор базы данных решит перенести файлы данных на другой диск для более равномерного распределения операций ввода/вывода по дискам, никаких проблем для приложения это не создаст. На работе приложения это никак не отразится.

Итак, иерархия объектов, обеспечивающих хранение данных в Oracle, выглядит так.

1. База данных, состоящая из одного или нескольких табличных пространств.
2. Табличное пространство, состоящее из одного или нескольких файлов данных. Табличное пространство содержит сегменты.
3. Сегмент **(TABLE, INDEX** и т.д.), состоящий из одного и более экстентов. Сегмент привязан к табличному пространству, но его данные могут находиться в разных файлах данных, образующих это табличное пространство.
4. Экстент — набор расположенных рядом на диске блоков. Экстент целиком находится в одном табличном пространстве и, более того, в одном файле данных этого табличного пространства.
5. Блок — наименьшая единица управления пространством в базе данных. Блок — наименьшая единица ввода/вывода, используемая сервером.

Прежде чем завершить описание файлов данных, давайте разберемся, как происходит управление экстентами в табличном пространстве. До версии 8.1.5 в Oracle был только один метод управления выделением экстентов в табличном пространстве. Этот метод называется *управление табличным пространством по словарю*. Т.е. место в табличном пространстве отслеживается в таблицах словаря данных (аналогично тому, как отслеживаются движения средств на банковских счетах с помощью пары таблиц **DEBIT** и **CREDIT).** В качестве дебета можно рассматривать выделенные объектам экстенты, в качестве кре-

дита — свободные для использования экстенты. Когда для объекта необходим очередной экстент, он запрашивается у системы. При получении такого запроса сервер Oracle обращается к соответствующим таблицам словаря данных, выполняет ряд запросов, находит (или не находит) свободное место нужного размера, а затем изменяет строку в одной таблице (или удаляет ее) и вставляет строку в другую. При этом сервер Oracle управляет пространством примерно так же, как работают обычные приложения: он изменяет данные в таблицах.

Соответствующие SQL-операторы для получения дополнительного пространства, выполняемые в фоновом режиме от имени пользователя, называют *рекурсивными.* Выполненный пользователем SQL-оператор INSERT вызывает выполнение других рекурсивных SQL-операторов для получения пространства на диске. При частом выполнении рекурсивные SQL-операторы создают весьма существенную дополнительную нагрузку на систему. Подобные изменения словаря данных должны выполняться последовательно, делать их одновременно нельзя. По возможности их следует избегать.

В прежних версиях Oracle этот метод управления пространством (и связанные с ним дополнительные расходы ресурсов на выполнение рекурсивных SQL-операторов) приводил к проблемам при работе с временными табличными пространствами (до появления "настоящих" временных табличных пространств). Речь идет о табличных пространствах, в которых необходимо часто выделять место (при этом надо удалить строку из одной таблицы словаря данных и вставить в другую) и освобождать его (помещая только что перенесенные строки туда, где они ранее были). Эти операции выполняются последовательно, что существенно снижает возможности одновременной работы и увеличивает время ожидания. В версии 7.3 СУБД Oracle для решения этой проблемы добавили временные пространства. Во временном табличном пространстве пользователь не мог создавать постоянные объекты. Это было единственным новшеством: управление пространством все равно выполнялось с помощью таблиц словаря данных. Однако после выделения экстента во временном табличном пространстве система его уже не освобождала. При следующем запросе экстента из временного табличного пространства сервер Oracle искал уже выделенный экстент в соответствующей структуре данных в памяти и, найдя, использовал повторно. В противном случае экстент выделялся как обычно. При этом после запуска и работы СУБД в течение некоторого времени соответствующий временный сегмент выглядел как заполненный, но фактически был просто "выделен". Свободными экстентами в нем управляли по-другому. При запросе сеансом временного пространства сервер Oracle искал его в структурах данных в памяти, а не выполнял дорогостоящие рекурсивные SQL-операторы.

В версиях Oracle, начиная с 8.1.5, был сделан следующий шаг по сокращению расхода ресурсов на управление пространством. Кроме табличных пространств, управляемых по словарю данных, появились *локально управляемые табличные пространства.* Для всех табличных пространств стало можно делать то, что в Oracle7.3 делалось для временных, т.е. не использовать словарь данных для управления свободным местом в табличном пространстве. В локально управляемом табличном пространстве для отслеживания экстентов используется битовая карта, хранящаяся в каждом файле данных. Теперь для получения экстента достаточно установить значение 1 для соответствующего бита в битовой карте. Для освобождения экстента — сбросить бит обратно в 0. По сравнению

с обращениями к словарю данных, это выполняется молниеносно. Больше не требуется ждать завершения длительно выполняемой операции, на уровне базы данных последовательно выделяющей место во всех табличных пространствах. Очередность на уровне табличного пространства остается только для очень быстро выполняемой операции. Локально управляемые табличные пространства имеют и другие положительные качества, например устанавливают одинаковый размер всех экстентов, но это имеет значение только для администраторов баз данных.

Временные файлы

Временные файлы данных в Oracle — это специальный тип файлов данных. Сервер Oracle использует временные файлы для хранения промежуточных результатов сортировки большого объема данных или результирующих множеств, если для них не хватает оперативной памяти. Постоянные объекты данных, такие как таблицы или индексы, во временных файлах никогда не хранятся, в отличие от содержимого временных таблиц и построенных по ним индексов. Так что создать таблицы приложения во временном файле данных нельзя, а вот хранить в нем данные можно, если использовать временную таблицу.

Сервер Oracle обрабатывает временные файлы специальным образом. Обычно все изменения объектов записываются в журналы повторного выполнения. Эти журналы транзакций в дальнейшем можно использовать для повторного выполнения транзакций. Это делается, например, при восстановлении после сбоя. Временные файлы в этом процессе не участвуют. Для них не генерируются данные повторного выполнения, хотя и генерируются данные отмены (UNDO) при работе с глобальными временными таблицами, чтобы можно было откатить изменения, сделанные в ходе сеанса. Создавать резервные копии временных файлов данных нет необходимости, а если кто-то это делает, то напрасно теряет время, поскольку данные во временном файле данных восстановить все равно нельзя.

Рекомендуется конфигурировать базу данных так, чтобы временные табличные пространства управлялись локально. Убедитесь, что ваш администратор базы данных использует оператор CREATE TEMPORARY TABLESPACE. Никому не нужно еще одно обычное табличное пространство, используемое под временные данные, поскольку не удастся получить преимущества временных файлов данных. Убедитесь также, что в качестве временного используется локально управляемое табличное пространство с экстентами одинакового размера, соответствующего значению параметра инициализации sort_area_size. Создаются такие временные табличные пространства примерно так:

```
tkyte@TKYTE816> create temporary tablespace temp
  2   tempfile  'c:\oracle\oradata\tkyte816\temp.dbf'
  3   size 5m
  4   extent management local
  5   uniform size 64k;

Tablespace created.
```

Поскольку мы опять вторгаемся в сферу деятельности администратора базы данных (АБД), переходим к следующей теме.

Управляющие файлы

Управляющий файл — это сравнительно небольшой файл (в редких случаях он может увеличиваться до 64 Мбайт), содержащий информацию обо всех файлах, необходимых серверу Oracle. Из файла параметров инициализации (init.ora) экземпляр может узнать, где находятся управляющие файлы, а в управляющем файле описано местонахождение файлов данных и файлов журнала повторного выполнения. В управляющих файлах хранится и другие необходимые серверу **Oracle** сведения, в частности время обработки контрольных точек, имя базы данных (которое должно совпадать со значением параметра инициализации db_name), дата и время создания базы данных, хронология архивирования журналов повторного выполнения (именно она приводит к увеличению размера управляющего файла в некоторых случаях) и т.д.

Управляющие файлы надо мультиплексировать либо аппаратно (располагать на RAID-массиве), либо с помощью средств сервера Oracle, когда RAID-массив или зеркалирование использовать нельзя. Необходимо поддерживать несколько копий этих файлов, желательно на разных дисках, чтобы предотвратить потерю управляющих файлов в случае сбоя диска. Потеря управляющих файлов — не фатальное событие, она только существенно усложнит восстановление.

С управляющими файлами разработчику скорее всего сталкиваться никогда не придется. Для администратора базы данных это — важная часть базы данных, но для разработчика эти файлы не особенно нужны.

Файлы журнала повторного выполнения

Файлы журнала повторного выполнения принципиально важны для базы данных Oracle. Это журналы транзакций базы данных. Они используются только для восстановления при сбое экземпляра или носителя или при поддержке резервной базы данных на случай сбоев. Если на сервере, где работает СУБД, отключится питание и вследствие этого произойдет сбой экземпляра, для восстановления системы в состояние, непосредственно предшествующее отключению питания, сервер Oracle при повторном запуске будет использовать активные журналы повторного выполнения. Если диск, содержащий файлы данных, полностью выйдет из строя, для восстановления резервной копии этого диска на соответствующий момент времени сервер Oracle, помимо активных журналов повторного выполнения, будет использовать также архивные. Кроме того, при случайном удалении таблицы или какой-то принципиально важной информации, если эта операция зафиксирована, с помощью активных и заархивированных журналов повторного выполнения можно восстановить данные из резервной копии на момент времени, непосредственно предшествующий удалению.

Практически каждое действие, выполняемое в СУБД Oracle, генерирует определенные данные повторного выполнения, которую надо записать в активные файлы журнала повторного выполнения. При вставке строки в таблицу конечный результат этой операции записывается в журналы повторного выполнения. При удалении строки записывается факт удаления. При удалении таблицы в журнал повторного выполнения записываются последствия этого удаления. Данные из удаленной таблицы не записываются, но рекурсивные SQL-операторы, выполняемые сервером Oracle при удалении таб-

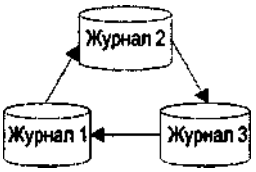
лицы, генерируют определенные данные повторного выполнения. Например, при этом сервер Oracle удалит строку из таблицы **SYS.OBJ$,** и это удаление будет отражено в журнале.

Некоторые операции могут выполняться в режиме с минимальной генерацией информации повторного выполнения. Например, можно создать индекс с атрибутом **NOLOGGING.** Это означает, что первоначальное создание этого индекса не будет записываться в журнал, но любые инициированные при этом рекурсивные SQL-операторы, выполняемые сервером Oracle, — будут. Например, вставка в таблицу **SYS.OBJ$** строки, соответствующей индексу, в журнал записываться не будет. Однако последующие изменения индекса при выполнении SQL-операторов **INSERT, UPDATE** и **DELETE,** будут записываться в журнал.

Есть два типа файлов журнала повторного выполнения: активные и архивные. В главе 5 мы еще раз затронем тему журналов повторного выполнения и сегментов отката, чтобы понять, как они влияют на разработку приложений. Пока же мы опишем, что собой представляют журналы повторного выполнения и их назначение.

Активный журнал повторного выполнения

В каждой базе данных Oracle есть как минимум два активных файла журнала повторного выполнения. Эти активные файлы журнала повторного выполнения имеют фиксированный размер и используются циклически. Сервер Oracle выполняет запись в файл журнала 1, а когда доходит до конца этого файла, — переключается на файл журнала 2 и переписывает его содержимое от начала до конца. Когда заполнен файл журнала 2, сервер переключается снова на файл журнала 1 (если имеется всего два файла журнала повторного выполнения; если их три, сервер, разумеется, переключится на третий файл):



Переход с одного файла журнала на другой называется *переключением журнала.* Важно отметить, что переключение журнала может вызвать временное "зависание" плохо настроенной базы данных. Поскольку журналы повторного выполнения используются для восстановления транзакций в случае сбоя, перед повторным использованием файла журнала необходимо убедиться, что его содержимое не понадобится в случае сбоя. Если сервер Oracle "не уверен", что содержимое файла журнала не понадобится, он приостанавливает на время изменения в базе данных и убеждается, что данные, "защищаемые" этой информацией повторного выполнения, записаны на диск. После этого обработка возобновляется, и файл журнала переписывается. Мы затронули ключевое понятие баз данных — *обработку контрольной точки.* Чтобы понять, как используются активные журналы повторного выполнения, надо разобраться с обработкой контрольной точки, использованием буферного кэша базы данных и рассмотреть функции *процесса записи бло-*

ков базы данных (Database Block Writer — **DBWn).** Буферный кэш и процесс **DBWn** подробно рассматриваются ниже, но мы все равно забегаем вперед, так что имеет смысл поговорить о них.

В буферном кэше базы данных временно хранятся блоки базы данных. Это структура в области SGA разделяемой памяти экземпляра Oracle. При чтении блоки запоминаются в этом кэше (предполагается, что в дальнейшем их не придется читать с диска). Буферный кэш — первое и основное средство настройки производительности сервера. Он существует исключительно для ускорения очень медленного процесса ввода/вывода. При изменении блока путем обновления одной из его строк изменения выполняются в памяти, в блоках буферного кэша. Информация, достаточная для повторного выполнения этого изменения, записывается в буфер журнала повторного выполнения — еще одну структуру данных в области SGA. При фиксации изменений с помощью оператора **COMMIT** сервер Oracle не записывает на диск все измененные блоки в области SGA. Он только записывает в активные журналы повторного выполнения содержимое буфера журнала повторного выполнения. Пока измененный блок находится в кэше, а не на диске, содержимое соответствующего активного журнала может быть использовано в случае сбоя экземпляра. Если сразу после фиксации изменения отключится питание, содержимое буферного кэша пропадет.

Если это произойдет, единственная запись о выполненном изменении останется в файле журнала повторного выполнения. После перезапуска экземпляра сервер Oracle будет по сути повторно выполнять транзакцию, изменяя блок точно так же, как мы это делали ранее, и фиксируя это изменение автоматически. Итак, если измененный блок находится в кэше и не записан на диск, мы не можем повторно записывать соответствующий файл журнала повторного выполнения.

Тут и вступает в игру процесс **DBWn.** Это фоновый процесс сервера Oracle, отвечающий за освобождение буферного кэша при заполнении и обработку *контрольных точек*. Обработка контрольной точки состоит в сбросе грязных (измененных) блоков из буферного кэша на диск. Сервер Oracle делает это автоматически, в фоновом режиме. Обработка контрольной точки может быть вызвана многими событиями, но чаще всего — переключением журнала повторного выполнения. При заполнении файла журнала 1, перед переходом на файл журнала 2, сервер Oracle инициирует обработку контрольной точки. В этот момент процесс **DBWn** начинает сбрасывать на диск все грязные блоки, защищенные файлом журнала I. Пока процесс **DBWn** не сбросит все блоки, защищаемые этим файлом, сервер Oracle не сможет его повторно использовать. Если попытаться использовать его прежде, чем процесс **DBWn** завершит обработку контрольной точки, в *журнал сообщений* (alert log) будет выдано следующее сообщение:

```
Thread 1 cannot allocate new log, sequence 66
Checkpoint not complete
  Current log# 2 seq# 65 mem# 0: C:\ORACLE\ORADATA\TKYTE816\REDO02.LOG
```

Журнал сообщений — это файл на сервере, содержащий информационные сообщения сервера, например, о запуске и останове, а также уведомления об исключительных ситуациях, вроде незавершенной обработки контрольной точки. Итак, в момент выда-

чи этого сообщения обработка изменений была приостановлена до завершения процессом **DBWn** обработки контрольной точки. Для ускорения обработки сервер Oracle отдал все вычислительные мощности процессу **DBWn.**

При соответствующей настройке сервера это сообщение в журнале появляться не должно. Если оно все же есть, значит, имеют место искусственные, ненужные ожидания, которых можно избежать. Цель (в большей степени администратора базы данных, чем разработчика) — иметь достаточно активных файлов журнала повторного выполнения. Это предотвратит попытки сервера использовать файл журнала, прежде чем будет закончена обработка контрольной точки. Если это сообщение выдается часто, значит, администратор базы данных не выделил для приложения достаточного количества активных журналов повторного выполнения или процесс **DBWn** не настроен как следует. Разные приложения генерируют различные объемы информации повторного выполнения. Системы класса СППР (системы поддержки принятия решений, выполняющие только запросы), естественно, будут генерировать намного меньше информации повторного выполнения, чем системы ООТ (системы оперативной обработки транзакций). Система, манипулирующая изображениями в больших двоичных объектах базы данных, может генерировать во много раз больше данных повторного выполнения, чем простая система ввода заказов. В системе ввода заказов со 100 пользователями генерируется в десять раз меньше данных повторного выполнения, чем в системе с 1000 пользователей. "Правильного" размера для журналов повторного выполнения нет, — он просто должен быть достаточным.

При определении размера и количества активных журналов повторного выполнения необходимо учитывать много факторов. Они, в общем, выходят за рамки книги, но я перечислю хотя бы отдельные, чтобы вы поняли, о чем речь.

- **Резервная база данных.** Когда заполненные журналы повторного выполнения посылаются на другую машину и там применяются к копии текущей базы данных, необходимо много небольших файлов журнала. Это поможет уменьшить рассинхронизацию резервной базы данных с основной.
- **Множество пользователей, изменяющих одни и те же блоки.** Здесь могут понадобиться большие файлы журнала повторного выполнения. Поскольку все изменяют одни и те же блоки, желательно, чтобы до того как блоки будут сброшены на диск, было выполнено как можно больше изменений. Каждое переключение журнала инициирует обработку контрольной точки, так что желательно переключать журналы как можно реже. Это, однако, может замедлить восстановление.
- **Среднее время восстановления.** Если необходимо обеспечить максимально быстрое восстановление, придется использовать файлы журнала меньшего размера, даже если одни и те же блоки изменяются множеством пользователей. Один или два небольших файла журнала повторного выполнения будут обработаны при восстановлении намного быстрее, чем один гигантский. Система в целом будет работать медленнее, чем могла бы (из-за слишком частой обработки контрольных точек), но восстановление будет выполняться быстрее. Для сокращения времени восстановления можно изменять и другие параметры базы данных, а не только уменьшать размер файлов журнала повторного выполнения.

Архивный журнал повторного выполнения

База данных Oracle может работать в двух режимах — NOARCHIVELOG и ARCHIVELOG. Я считаю, что система, используемая в производственных условиях, обязательно должна работать в режиме ARCHIVELOG. Если база данных не работает в режиме ARCHIVELOG, данные рано или поздно будут потеряны. Работать в режиме NOARCHIVELOG можно только в среде разработки или тестирования.

Эти режимы отличаются тем, что происходит с файлом журнала повторного выполнения до того как сервер Oracle его перепишет. Сохранять ли копию данных повторного выполнения или разрешить серверу Oracle переписать ее, потеряв при этом навсегда? — очень важный вопрос. Если не сохранить этот файл, мы не сможем восстановить данные с резервной копии до текущего момента. Предположим, резервное копирование выполняется раз в неделю, по субботам. В пятницу вечером, после того как за неделю было сгенерировано несколько сотен журналов повторного выполнения, происходит сбой диска. Если база данных не работала в режиме ARCHIVELOG, остается только два варианта дальнейших действий.

- Удалить табличное пространство/пространства, связанные со сбойным диском. Любое табличное пространство, имеющее файлы данных на этом диске, должно быть удалено, включая его содержимое. Если затронуто табличное пространство SYSTEM (словарь данных Oracle), этого сделать нельзя.
- Восстановить данные за субботу и потерять все изменения за неделю.

Оба варианта непривлекательны, поскольку приводят к потере данных. Работая же в режиме ARCHIVELOG, достаточно найти другой диск и восстановить на него соответствующие файлы с субботней резервной копии. Затем применить к ним архивные журналы повторного выполнения и, наконец, — активные журналы повторного выполнения (то есть повторить все накопленные за неделю транзакции в режиме быстрого наката). При этом ничего не теряется. Данные восстанавливаются на момент сбоя.

Часто приходится слышать, что в производственных системах режим ARCHIVELOG не нужен. Это глубочайшее заблуждение. Если не хотите в один момент потерять данные, сервер должен работать в режиме ARCHIVELOG. "Мы используем дисковый массив RAID-5 и абсолютно защищены" — вот типичное оправдание. Я сталкивался с ситуациями, когда по вине изготовителя все пять дисков массива одновременно останавливались. Я видел поврежденные аппаратным контроллером файлы данных, которые в поврежденном виде надежно защищались дисковым массивом. Если имеется резервная копия данных на момент, предшествующий сбою оборудования, и архивы не повреждены, — восстановление возможно. Поэтому нет разумных оснований для того, чтобы не использовать режим ARCHIVELOG в системе, где данные представляют хоть какую-нибудь ценность. Производительность — не основание. При правильной настройке на архивирование расходуется незначительное количество ресурсов системы. Это, а также тот факт, что быстро работающая система, в которой данные теряются, — бесполезна, заставляет сделать вывод, что, даже если бы архивирование журналов замедляло работу системы в два раза, оно в любом случае должно выполняться.

Не поддавайтесь ни на какие уговоры и не отказывайтесь от режима **ARCHIVELOG.** Вы потратили много времени на разработку приложения, поэтому надо, чтобы пользователи ему доверяли. О доверии можно забыть, если их данные будут потеряны.

Итак, мы рассмотрели основные типы файлов, используемых сервером Oracle: от небольших файлов параметров инициализации (без которых не удастся даже запустить экземпляр) до принципиально важных файлов журнала повторного выполнения и файлов данных. Обсудили структуры хранения данных в Oracle: от табличных пространств, сегментов и экстентов до блоков базы данных — наименьшей единицы хранения. Была описана обработка контрольной точки в базе данных и даже (несколько преждевременно) затронута работа физических процессов или потоков, составляющих экземпляр Oracle. В последующих разделах этой главы мы более детально рассмотрим эти процессы и структуры памяти.

Структуры памяти

Теперь пришло время рассмотреть основные структуры памяти сервера Oracle. Их три.

- SGA, System Global Area — глобальная область системы. Это большой совместно используемый сегмент памяти, к которому обращаются все процессы Oracle.
- **PGA,** Process Global Area — глобальная область процесса. Это приватная область памяти процесса или потока, недоступная другим процессам/потокам.
- **UGA,** User Global Area — глобальная область пользователя. Это область памяти, связанная с сеансом. Глобальная область памяти может находиться в **SGA** либо в **PGA.** Если сервер работает в режиме MTS, она располагается в области **SGA,** если в режиме выделенного сервера, — в области **PGA.**

Рассмотрим кратко области **PGA** и UGA, затем перейдем к действительно большой структуре — области **SGA.**

Области PGA и UGA

Как уже было сказано, **PGA** — это область памяти процесса. Эта область памяти используется одним процессом или одним потоком. Она недоступна ни одному из остальных процессов/потоков в системе. Область **PGA** обычно выделяется с помощью библиотечного вызова **malloc()** языка С и со временем может расти (или уменьшаться). Область **PGA** никогда не входит в состав области **SGA** — она всегда локально выделяется процессом или потоком.

Область памяти UGA хранит состояние сеанса, поэтому всегда должна быть ему доступна. Местонахождение области UGA зависит исключительно от конфигурации сервера Oracle. Если сконфигурирован режим MTS, область **UGA** должна находиться в структуре памяти, доступной всем процессам, следовательно, в **SGA.** В этом случае сеанс сможет использовать любой разделяемый сервер, так как каждый из них сможет прочитать и записать данные сеанса. При подключении к выделенному серверу это требование общего доступа к информации о состоянии сеанса снимается, и область UGA ста-

новится почти синонимом PGA, — именно там информация о состоянии сеанса и будет располагаться. Просматривая статистическую информацию о системе, можно обнаружить, что при работе в режиме выделенного сервера область UGA входит в **PGA** (размер области **PGA** будет больше или равен размеру используемой памяти **UGA** — размер UGA будет учитываться при определении размера области PGA).

Размер области **PGA/UGA** определяют параметры уровня сеанса в файле **init.ora: SORT_AREA_SIZE** и **SORT_AREA_RETAINED_SIZE.** Эти два параметра управляют объемом пространства, используемым сервером Oracle для сортировки данных перед сбросом на диск, и определяют объем сегмента памяти, который не будет освобожден по завершении сортировки. **SORT_AREA_SIZE** обычно выделяется в области **PGA,** а **SORT_AREA_RETAINED_SIZE** - в **UGA.** Управлять размером областей **UGA/PGA** можно с помощью запроса к специальному представлению **V$** сервера Oracle. Эти представления называют также представлениями динамической производительности. Подробнее эти представления **V$** рассматриваются в главе 10. С помощью представлений **V$** можно определить текущее использование памяти под области **PGA** и **UGA.** Например, запущен небольшой тестовый пример, требующий сортировки большого объема данных. Просмотрев несколько первых строк данных, я решил не извлекать остальное результирующее множество. После этого можно сравнить использование памяти "до" и "после":

```
tkyte@TKYTE816> select a.name, b.value
  2  from v$statname a, v$mystat b
  3  where a.statistic# = b.statistic#
  4  and a.name like '%ga %'
  5  /

NAME                                VALUE

session uga memory                  67532
session uga memory max              71972
session pga memory                 144688
session pga memory max             144688

4 rows selected.
```

Итак, перед началом сортировки в области **UGA** было около 70 Кбайт данных, а в **PGA** — порядка **140** Кбайт. Первый вопрос: сколько памяти используется в области **PGA** помимо UGA? Вопрос нетривиальный и на него нельзя ответить, не зная, подключен ли сеанс к выделенному или к разделяемому серверу; но даже зная это нельзя ответить однозначно. В режиме выделенного сервера область **UGA** входит в состав **PGA.** В этом случае порядка 140 Кбайт выделено в области памяти процесса или потока. В режиме MTS область UGA выделяется из **SGA,** а область **PGA** относится к разделяемому серверу. Поэтому при работе в режиме MTS к моменту получения последней строки из представленного выше запроса разделяемый серверный процесс уже может использоваться другим сеансом. Соответственно, область **PGA** уже не принадлежит нам, так что мы используем всего 70 Кбайт памяти (если только не находимся в процессе выполнения запроса, когда областями **PGA** и **UGA** суммарно используется 210 Кбайт памяти).

Теперь разберемся, что происходит в областях PGA/UGA нашего сеанса:

```
tkyte@TKYTE816> show parameter sort_area

NAME                                 TYPE     VALUE

sort_area_retained_size              integer  65536
sort_area_size                       integer  65536

tkyte@TKYTE816> set pagesize 10
tkyte@TKYTE816> set pause on
tkyte@TKYTE816> select * from all_objects order by 1, 2, 3, 4;


...(Нажмите Control-C после первой страницы данных)  ...

tkyte@TKYTE816> set pause off

tkyte@TKYTE816> select a.name, b.value
  2  from v$statname a, v$mystat b
  3  where a.statistic# = b.statistic#
  4  and a.name like '%ga %'
5  /

NAME                                  VALUE

session uga memory                     67524
session uga memory max                174968
session pga memory                    291336
session pga memory max                291336

4 rows selected.
```

Как видите, памяти использовано больше, поскольку данные сортировались. Область **UGA** временно увеличилась примерно на размер **SORT_AREA_RETAINED_SIZE,** а область **PGA** — немного больше. Для выполнения запроса и сортировки сервер Oracle выделил дополнительные структуры, которые оставлены в памяти сеанса для других запросов. Давайте выполним ту же операцию, изменив значение **SORT_AREA_SIZE:**

```
tkyte@TKYTE816> alter session set sort_area_size=1000000;

Session altered.

tkyte@TKYTE816> select a.name, b.value
  2  from v$statname a, v$mystat b
  3  where a.statistic# = b.statistic#
  4  and a.name like '%ga %'
  5  /

NAME                                  VALUE

session uga memory                     63288
session uga memory max                174968
session pga memory                    291336
session pga memory max                291336
```

```
4 rows selected.

tkyte@TKYTE816> show parameter sort_area

NAME                                 TYPE        VALUE

sort_area_retained_size              integer     65536

sort_area_size                       integer     1000000

tkyte@TKYTE816> select * from all_objects order by 1, 2, 3, 4;

...(Нажмите Control-C после первой страницы данных) ...

tkyte@TKYTE816> set pause off

tkyte@TKYTE816> select a.name, b.value
  2  from v$statname a, v$mystat b
  3  where a.statistic# = b.statistic#
  4  and a.name like '%ga %'
  5  /
NAME                                     VALUE

session uga memory                       67528
session uga memory max                  174968
session pga memory                     1307580
session pga memory max                 1307580

4 rows selected.
```

Как видите, в этот раз область PGA увеличилась существенно. Примерно на 1000000 байт, в соответствии с заданным значением SORT_AREA_SIZE. Интересно отметить, что в этот раз размер области UGA вообще не изменился. Для ее изменения надо задать другое значение **SORT_AREA_RETAINED_SIZE**, как показано ниже:

```
tkyte@TKYTE816> alter session set sort_area_retained_size=1000000;
Session altered.

tkyte@TKYTE816> select a.name, b.value
  2  from v$statname a, v$mystat b
  3  where a.statistic# = b.statistic#
  4  and a.name like '%ga %'
  5  /

NAME                                     VALUE

session uga memory                       63288
session uga memory max                  174968
session pga memory                     1307580
session pga memory max                 1307580

4 rows selected.
```

```
tkyte@TKYTE816> show parameter sort_area

KAME                                 TYPE     VALUE

sort_area_retained_size              integer  1000000
sort_area_size                       integer  1000000

tkyte@TKYTE816> select * from all_objects order by 1, 2, 3, 4;


...(Нажмите Control-C после первой страницы данных)  ...

tkyte@TKYTE816> select a.name, b.value
  2   from v$statname a, v$mystat b
  3   where a.statistic# * b.statistic#
  4   and a.name like '%ga %'
  5   /

NAME                                   VALUE

session uga memory                     66344
session uga memory max               1086120
session pga memory                   1469192
session pga memory max               1469192

4 rows selected.
```

Теперь мы видим, что существенное увеличение размера области **UGA** связано с необходимостью дополнительно принять данные размером **SORT_AREA_RETAINED_SIZE.** В ходе обработки запроса 1 Мбайт сортируемых данных "кэширован в памяти". Остальные данные были на диске (где-то во временном сегменте). По завершении выполнения запроса это дисковое пространство возвращено для использования другими сеансами. Обратите внимание, что область **PGA** не уменьшилась до прежнего размера. Этого следовало ожидать, поскольку область **PGA** используется как "куча" и состоит из фрагментов, выделенных с помощью вызовов **malloc().** Некоторые процессы в сервере Oracle явно освобождают память **PGA,** другие же оставляют выделенную память в куче (область для сортировки, например, остается в куче). Сжатие кучи при этом обычно ничего не дает (размер используемой процессами памяти только растет). Поскольку область UGA является своего рода "подкучей" (ее "родительской" кучей является область **PGA** либо SGA), она может сжиматься. При необходимости можно принудительно сжать область **PGA:**

```
tkyte@TKYTE816> exec dbms_session.free_unused_user_memory;

PL/SQL procedure successfully completed.


tkyte@TKYTE816> select a.name, b.value
  2   from v$statname a, v$mystat b
.  3   where a.statistic# = b.statistic#
  4   and a.name like '%ga %'
  5   /
```

```
KAME                                VALUE

session uga memory                   73748
session uga memory max              1086120
session pga memory                   183360
session pga memory max              1469192
```

Учтите, однако, что в большинстве систем это действие — пустая трата времени. Можно уменьшить размер кучи **PGA** в рамках экземпляра Oracle, но память при этом операционной системе не возвращается. В зависимости от принятого в ОС метода управления памятью, суммарное количество используемой памяти даже увеличится. Все зависит от того, как на данной платформе реализованы функции **malloc(), free(), realloc(), brk()** и **sbrk()** (стандартные функции управления памятью в языке C).

Итак, мы рассмотрели две структуры памяти, области **PGA** и **UGA.** Теперь понятно, что область **PGA** принадлежит процессу. Она представляет собой набор переменных, необходимых выделенному или разделяемому серверному процессу Oracle для поддержки сеанса. Область **PGA** — это "куча" памяти, в которой могут выделяться другие структуры. Область **UGA** также является кучей, в которой определяются связанные с сеансом структуры. Область **UGA** выделяется из **PGA** при подключении к выделенному серверу Oracle и — из области **SGA** при подключении в режиме MTS. Это означает, что при работе в режиме MTS необходимо задать такой размер области **SGA,** чтобы в ней хватило места под области **UGA** для предполагаемого максимального количества одновременно подключенных к базе данных пользователей. Поэтому область **SGA** в экземпляре, работающем в режиме MTS, обычно намного больше, чем область **SGA** аналогично сконфигурированного экземпляра, работающего в режиме выделенного сервера.

Область SGA

Каждый экземпляр Oracle имеет одну большую область памяти, которая называется SGA, System Global Area — глобальная область системы. Это большая разделяемая структура, к которой обращаются все процессы Oracle. Ее размер варьируется от нескольких мегабайт в небольших тестовых системах, до сотен мегабайт в системах среднего размера и множества гигабайт в больших системах.

В ОС UNIX область SGA — это физический объект, которую можно "увидеть" с помощью утилит командной строки. Физически область **SGA** реализована как сегмент разделяемой памяти — отдельный фрагмент памяти, к которому могут подключаться процессы. При отсутствии процессов Oracle вполне допустимо иметь в системе область SGA; память существует отдельно от них. Однако наличие области **SGA** при отсутствии процессов Oracle означает, что произошел тот или иной сбой экземпляра. Эта ситуация — нештатная, но она бывает. Вот как "выглядит" область SGA в ОС UNIX:

```
$ ipcs -mb
IPC status from <running system> as of Mon Feb 19 14:48:26 EST 2001
T         ID        KEY        MODE         OWNER     GROUP       SEGSZ
Shared Memory:
m         105   0xf223dfc8 -rw-r----------ora816        dba   186802176
```

В ОС Windows увидеть область **SGA,** как в ОС UNIX, нельзя. Поскольку на этой платформе экземпляр Oracle работает как единый процесс с одним адресным пространством, область **SGA** выделяется как приватная память процесса **ORACLE.EXE.** С помощью диспетчера задач Windows (Task Manager) или другого средства контроля производительности можно узнать, сколько памяти выделено процессу **ORACLE.EXE,** но нельзя определить, какую часть по отношению к другим выделенным структурам памяти занимает область **SGA.**

В самой СУБД Oracle можно определить размер области **SGA** независимо от платформы. Есть еще одно "магическое" представление V$, именуемое **V$SGASTAT.** Вот как его можно использовать:

```
tkyte@TKYTE816> compute sum of bytes on pool
tkyte@TKYTE816> break on pool skip 1
tkyte@TKYTE816> select pool, name, bytes
  2   from v$sgastat
  3   order by pool, name;
```

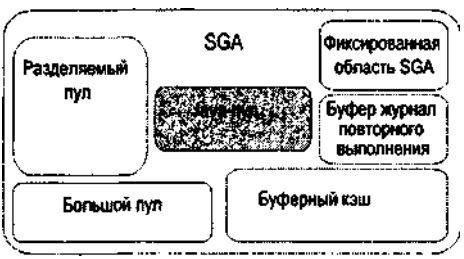
POOL	NAME	BYTES
Java pool	free memory	18366464
	memory in use	2605056
sum		20971520
large pool	free memory	6079520
	session heap	64480
sum		6144000
shared pool	Checkpoint queue	73764
	KGFF heap	5900
	KGK heap	17556
	KQLS heap	554560
	PL/SQL DIANA	364292
	PL/SQL MPCODE	138396
	PLS non-lib hp	2096
	SYSTEM PARAMETERS	61856
	State objects	125464
	VIRTUAL CIRCUITS	97752
	character set object	58936
	db_block_buffers	408000
	db_block_hash_buckets	179128
	db_files	370988
	dictionary cache	319604
	distributed_transactions	180152
	dlo fib struct	40980
	enqueue resources	94176
	event statistics per sess	201600
	file * translation table	65572
	fixed allocation callback	320
	free memory	9973964
	joxlod: in ehe	52556

```
                 joxlod: in phe                    4144
                 joxs heap init                     356
                 library cache                  1403012
                 message pool freequeue          231152
                 miscellaneous                   562744
                 processes                        40000
                 sessions                        127920
                 sql area                       2115092
                 table columns                    19812
                 transaction branches            368000
                 transactions                     58872
                 trigger defini                    2792
                 trigger inform                     520
-------------------------------------------------------
sum                                             18322028

                 db_block_buffers               24576000
                 fixed_sga                         70924
                 log buffer                        66560
**********
-------------------------------------------------------
sum                                             24713484

43 rows selected.
```

Область **SGA** разбита на несколько пулов.

- **Java-пул.** Java-пул представляет собой фиксированный пул памяти, выделенный виртуальной машине JVM, которая работает в составе сервера.
- **Большой пул.** *Большой пул* (large pool) используется сервером в режиме MTS для размещения памяти сеанса, средствами распараллеливания Parallel Execution для буферов сообщений и при резервном копировании с помощью RMAN для буферов дискового ввода/вывода.
- **Разделяемый** пул. *Разделяемый пул* (shared pool) содержит разделяемые курсоры, хранимые процедуры, объекты состояния, кэш словаря данных и десятки других компонентов данных.
- **"Неопределенный" ("Null") пул.** Этот пул не имеет имени. Это память, выделенная под буферы блоков (кэш блоков базы данных, буферный кэш), буфер журнала повторного выполнения и "фиксированную область SGA".

Поэтому графически область **SGA** можно представить следующим образом:



На общий размер **SGA** наиболее существенно влияют следующие параметры **init.ora.**

- **JAVA_POOL_SIZE.** Управляет размером Java-пула.
- **SHARED_POOL_SIZE.** Управляет (до некоторой степени) размером разделяемого пула.
- **LARGE_POOL_SIZE.** Управляет размером большого пула.
- **DB_BLOCK_BUFFERS.** Управляет размером буферного кэша.
- **LOG_BUFFER.** Управляет (отчасти) размером буфера журнала повторного выполнения.

За исключением параметров **SHARED_POOL_SIZE** и **LOG_BUFFER,** имеется однозначное соответствие между значением параметров в файле **init.ora** и объемом памяти, выделяемой соответствующей структуре в области **SGA.** Например, если умножить **DB_BLOCK_BUFFERS** на размер буфера, получится значение, совпадающее с размером в строке **DB_BLOCK_BUFFERS** для пула **NULL** в представлении **V$SGASTAT** (добавляется определенный объем памяти на поддержку защелок). Суммарное количество байтов, вычисленное из строк представления **V$SGASTAT** для большого пула, совпадет со значением параметра инициализации **LARGE_POOL_SIZE.**

Фиксированная область SGA

Фиксированная область **SGA** — это часть области **SGA,** размер которой зависит от платформы и версии. Она "компилируется" в двоичный модуль сервера Oracle при установке (отсюда и название — "фиксированная"). Фиксированная область **SGA** содержит переменные, которые указывают на другие части **SGA,** а также переменные, содержащие значения различных параметров. Размером фиксированной области **SGA** (как правило, очень небольшой) управлять нельзя. Можно рассматривать эту область как "загрузочную" часть **SGA,** используемую сервером Oracle для поиска других компонентов SGA.

Буфер журнала повторного выполнения

Буфер журнала повторного выполнения используется для временного кэширования данных активного журнала повторного выполнения перед записью на диск. Поскольку перенос данных из памяти в память намного быстрее, чем из памяти — на диск, использование буфера журнала повторного выполнения позволяет существенно ускорить работу сервера. Данные не задерживаются в буфере журнала повторного выполнения надолго. Содержимое этого буфера сбрасывается на диск:

- раз в три секунды;
- при фиксации транзакции;
- при заполнении буфера на треть или когда в нем оказывается 1 Мбайт данных журнала повторного выполнения.

Поэтому создание буфера журнала повторного выполнения размером в десятки Мбайт — напрасное расходование памяти. Чтобы использовать буферный кэш журнала повторного выполнения размером 6 Мбайт, например, надо выполнять продолжительные тран-

закции, генерирующие по 2 Мбайт информации повторного выполнения не более чем за три секунды. Если кто-либо в системе зафиксирует транзакцию в течение этих трех секунд, в буфере не будет использовано и 2 Мбайт, — содержимое буфера будет регулярно сбрасываться на диск. Лишь очень немногие приложения выиграют от использования буфера журнала повторного выполнения размером в несколько мегабайт.

Стандартный размер буфера журнала повторного выполнения, задаваемый параметром LOG_BUFFER в файле **init.ora,** определяется как максимальное из значений 512 и (128 * количество процессоров) Кбайт. Минимальный размер этой области равен максимальному размеру блока базы данных для соответствующей платформы, умноженному на четыре. Если необходимо узнать это значение, установите **LOG_BUFFER** равным 1 байт и перезапустите сервер. Например, на моем сервере под Windows 2000 я получил следующий результат:

```
SVRMGR> show parameter log_buffer
NAME                            TYPE     VALUE

log_buffer                      integer  1

SVRMGR> select * from v$sgastat where name = 'log_buffer';
POOL     NAME                   BYTES

         log_buffer             66560
```

Теоретически минимальный размер буфера журнала повторного выполнения, независимо от установок в файле **init.ora,** в данном случае — 65 Кбайт. Фактически он немного больше:

```
tkyte@TKYTE816> select * from v$sga where name = 'Redo Buffers';

NAME                            VALUE

Redo Buffers                    77824
```

То есть размер буфера — 76 Кбайт. Дополнительное пространство выделено из соображений безопасности, как "резервные" страницы, защищающие страницы буфера журнала повторного выполнения.

Буферный кэш

До сих пор мы рассматривали небольшие компоненты области SGA. Теперь переходим к составляющей, которая достигает огромных размеров. В буферном кэше сервер Oracle хранит блоки базы данных перед их записью на диск, а также после считывания с диска. Это принципиально важный компонент **SGA.** Если сделать его слишком маленьким, запросы будут выполняться годами. Если же он будет чрезмерно большим, пострадают другие процессы (например, выделенному серверу не хватит пространства для создания области **PGA,** и он просто не запустится).

Блоки в буферном кэше контролируются двумя списками. Это список "грязных" блоков, которые должны быть записаны процессом записи блоков базы данных (это **DBWn;** его мы рассмотрим несколько позже). Есть еще список "чистых" блоков, организованный в Oracle 8.0 и предыдущих версиях в виде очереди (LRU — Least Recently Used). Блоки упорядочивались по времени последнего использования. Этот алгоритм был не-

много изменен в Oracle 8i и последующих версиях. Вместо физического упорядочения списка блоков, сервер Oracle с помощью счетчика, связанного с блоком, подсчитывает *количество обращений* ("touch count) при каждом *обращении* (hit) к этому блоку в буферном кэше. Это можно увидеть в одной из действительно "магических" таблиц X$. Эти таблицы не описаны в документации Oracle, но информация о них периодически просачивается.

Таблица **X$BH** содержит информацию о блоках в буферном кэше. В ней можно увидеть, как "счетчик обращений" увеличивается при каждом обращении к блоку. Сначала необходимо найти блок. Мы используем блок таблицы DUAL — специальной таблицы, состоящей из одной строки и одного столбца, которая есть во всех базах данных Oracle. Необходимо найти соответствующий номер файла и номер блока в файле:

```
tkyte@TKYTE816> select file_id, block_id
  2   from dba_extents
  3   where segment_name = 'DUAL' and owner = 'SYS';

  FILE_ID    BLOCK_ID

        1         465
```

Теперь можно использовать эту информацию для получения "счетчика обращений" для этого блока:

```
sys@TKYTE816> select tcn from x$bh where  file#=1  and dablk=465;

      TCH

       10

sys@TKYTE816> select * from dual;

D

X

sys@TKYTE816> select tch from x$bh where file#=1 and dablk=465;

      TCH

       11

sys@TKYTE816> select * from dual;

D

X

sys@TKYTE816> select tch from x$bh where file# = 1 and dablk = 465;

      TCH

       12
```

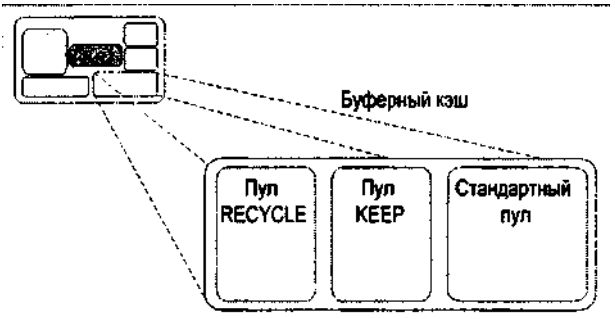
При каждом обращении к блоку увеличивается значение счетчика. Использованный буфер больше не переносится в начало списка. Он остается на месте, а его "счетчик

обращений" увеличивается. Блоки со временем перемешаются по списку естественным путем, поскольку измененные блоки переносятся в список "грязных" (для записи на диск процессом DBWn). Кроме того, если несмотря на повторное использование блоков буферный кэш заполнился, и блок с небольшим значением "счетчика обращений" удаляется из списка, он возвращается с новыми данными примерно в середину списка. Полный алгоритм управления списка довольно сложный и меняется с каждой новой версией Oracle. Подробности его работы несущественны для разработчиков, достаточно помнить, что интенсивно используемые блоки кэшируются надолго, а редко используемые — долго в кэше не задерживаются.

Буферный кэш в версиях до Oracle 8.0 представлял собой один большой кэш. Все блоки кэшировались одинаково, никаких средств деления пространства буферного кэша на части не существовало. В Oracle 8.0 добавлена возможность создания буферных пулов. С ее помощью можно зарезервировать в буферном кэше место для сегментов (как вы помните, сегменты соответствуют таблицам, индексам и т.д.). Появилась возможность выделить место (буферный пул) достаточного размера для размещения целиком в памяти, например, таблиц-справочников". При чтении сервером Oracle блоков из этих таблиц они кэшируются в этом специальном пуле. Они будут конфликтовать за место в пуле только с другими помещаемыми в него сегментами. Остальные сегменты в системе будут "сражаться" за место в стандартном буферном пуле. При этом повышается вероятность их кэширования: они не выбрасываются из кэша как устаревшие при считывании других, не связанных с ними блоков. Буферный пул, обеспечивающий подобное кэширование, называется пулом KEEP. Блоками в пуле KEEP сервер управляет так же, как в обычном буферном кэше. Если блок используется часто, он остается в кэше; если к блоку некоторое время не обращались и в буферном пуле не осталось места, этот блок выбрасывается из пула как устаревший.

Можно выделить еще один буферный пул. Он называется пулом RECYCLE. В нем блоки выбрасываются иначе, чем в пуле KEEP. Пул KEEP предназначен для продолжительного кэширования "горячих" блоков. Из пула RECYCLE блок выбрасывается сразу после использования. Это эффективно в случае "больших" таблиц, которые читаются случайным образом. (Понятие "большая таблица" очень относительно; нет эталона для определения того, что считать "большим".) Если в течение разумного времени вероятность повторного считывания блока мала, нет смысла долго держать такой блок в кэше. Поэтому в пуле RECYCLE блоки регулярно перечитываются.

Итак, уточняя схему SGA, ее можно представить так:



Разделяемый пул

Разделяемый пул — один из наиболее важных фрагментов памяти в области SGA, особенно для обеспечения производительности и масштабируемости. Слишком маленький разделяемый пул может снизить производительность настолько, что система будет казаться зависшей. Слишком большой разделяемый пул может привести к такому же результату. Неправильное использование разделяемого пула грозит катастрофой.

Итак, что же такое разделяемый пул? В *разделяемом пуле* сервер Oracle кэширует различные "программные" данные. Здесь кэшируются результаты разбора запроса. Перед повторным разбором запроса сервер Oracle просматривает разделяемый пул в поисках готового результата. Выполняемый сеансом PL/SQL-код тоже кэшируется здесь, так что при следующем выполнении не придется снова читать его с диска. PL/SQL-код в разделяемом пуле не просто кэшируется, — появляется возможность его совместного использования сеансами. Если 1000 сеансов выполняют тот же код, загружается и совместно используется всеми сеансами лишь одна копия этого кода. Сервер Oracle хранит в разделяемом пуле параметры системы. Здесь же хранится кэш словаря данных, содержащий информацию об объектах базы данных. Короче, в разделяемом пуле хранится все, кроме продуктов питания.

Разделяемый пул состоит из множества маленьких (около 4 Кбайт) фрагментов памяти. Память в разделяемом пуле управляется по принципу давности использования (LRU). В этом отношении она похожа на буферный кэш: если фрагмент не используется, он теряется. Стандартный пакет DBMS_SHARED_POOL позволяет изменить это и принудительно закрепить объекты в разделяемом пуле. Это позволяет загрузить часто используемые процедуры и пакеты при запуске сервера и сделать так, чтобы они не выбрасывались из пула как устаревшие. Обычно, если в течение определенного периода времени фрагмент памяти в разделяемом пуле не использовался, он выбрасывается как устаревший. Даже PL/SQL-код, который может иметь весьма большой размер, управляется механизмом постраничного устаревания, так что при выполнении кода очень большого пакета необходимый код загружается в разделяемый пул небольшими фрагментами. Если в течение продолжительного времени он не используется, то в случае переполнения выбрасывается из разделяемого пула, а пространство выделяется для других объектов.

Самый простой способ поломать механизм разделяемого пула Oracle — не использовать связываемые переменные. Как было показано в главе 1, отказавшись от использования связываемых переменных, можно "поставить на колени" любую систему, поскольку:

- система будет тратить много процессорного времени на разбор запросов;
- система будет тратить очень много ресурсов на управление объектами в разделяемом пуле, т.к. не предусмотрено повторное использование планов выполнения запросов.

Если каждый переданный серверу Oracle запрос специфичен, с жестко заданными константами, это вступает в противоречие с назначением разделяемого пула. Разделяемый пул создавался для того, чтобы хранящиеся в нем планы выполнения запросов использовались многократно. Если каждый запрос — абсолютно новый и никогда ранее

не встречался, в результате кэширования только расходуются дополнительные ресурсы. Разделяемый пул начинает **снижать производительность.** Обычно эту проблему пытаются решить, увеличивая разделяемый пул, но в результате становится еще хуже. Разделяемый пул снова неизбежно заполняется, и его поддержка требует **больших** ресурсов, чем поддержка маленького разделяемого пула, поскольку при управлении большим заполненным пулом приходится выполнять больше действий, чем при управлении маленьким заполненным пулом.

Единственным решением этой проблемы является применение разделяемых операторов SQL, которые используются повторно. В главе 10 мы опишем параметр инициализации **CURSOR_SHARING,** который можно использовать для частичного решения подобных проблем, но наиболее эффективное решение — применять повторно используемые SQL-операторы. Даже самые большие из крупных систем требуют от 10000 до 20000 уникальных SQL-операторов. В большинстве систем используется лишь несколько сотен уникальных запросов.

Следующий практический пример показывает, насколько все осложняется при неправильном использовании разделяемого пула. Меня пригласили поработать над системой, стандартной процедурой обслуживания которой была остановка экземпляра каждую ночь для очистки области SGA и последующий перезапуск. Это приходилось делать, поскольку в течение дня в системе возникали проблемы, связанные с избыточной загрузкой процессора, и, если сервер работал больше одного дня, производительность начинала падать. Единственная причина этого была в том, что за период с 9 утра до 5 вечера они полностью заполняли разделяемый пул размером 1 Гбайт в области SGA общим размером 1,1 Гбайт. Да, именно так: 0,1 Гбайта было выделено под буферный кэш и другие компоненты, а 1 Гбайт — для кэширования запросов, которые никогда не выполнялись повторно. Систему приходилось перезапускать, потому что свободная память в разделяемом пуле исчерпывалась в течение одного дня. На поиск и удаление устаревших структур (особенно из такого большого разделяемого пула) расходовалось столько ресурсов, что производительность резко падала (хотя она и до этого была далека от оптимальной, ведь приходилось управлять разделяемым пулом в 1 Гбайт). Кроме того, пользователи этой системы постоянно требовали добавления новых процессоров, поскольку жесткий разбор SQL-операторов требовал больших вычислительных ресурсов. Когда, после внесения исправлений, в приложении стали использоваться связываемые переменные, удалось не только снизить требования к ресурсам машины (у них и так вычислительные мощности намного превышали необходимые), но и появилась возможность пересмотреть распределение памяти. Вместо разделяемого пула размером в 1 Гбайт оказалось достаточно выделить 100 Мбайт, причем за много недель непрерывной работы он не заполнился.

И последнее, что хотелось бы сказать о разделяемом пуле и параметре инициализации **SHARED_POOL_SIZE.** Нет никакой связи между результатами выполнения запроса:

```
sys@TKYTE816> select sum(bytes) from v$sgastat where pool = 'shared pool';
  SUM(BYTES)

    18322028
1 row selected.
```

и значением параметра инициализации SHARED_POOL_SIZE:

```
sys@TKYTE816> show parameter shared_pool_size
NAME     TYPE                                VALUE

shared_pool_size                  string    15360000
SVRMGR>
```

кроме того, что значение SUM(BYTES) FROM V$SGASTAT всегда больше, чем значение параметра SHARED_POOL_SIZE. В разделяемом пуле хранится много других структур, не охватываемых соответствующим параметром инициализации. Значение SHARED_POOL_SIZE обычно является основным, но не единственным фактором, определяющим размер разделяемого пула SUM(BYTES). Например, параметр инициализации CONTROL_FILES задает управляющие файлы, а для каждого управляющего файла в разделе "прочее" разделяемого пула требуется 264 байта. Жаль, что показатель 'shared pool' в представлении V$SGASTAT и параметр инициализации SHARED_POOL_SIZE получили похожие названия, поскольку параметр инициализации влияет на размер разделяемого пула, но не задает его полностью.

Большой пул

Большой пул назван так не потому, что это "большая" структура (хотя его размер вполне может быть большим), а потому, что используется для выделения больших фрагментов памяти — больших, чем те, для управления которыми создавался разделяемый пул. До его появления в Oracle 8.0, выделение памяти выполнялось в рамках разделяемого пула. Это было неэффективно при использовании средств, выделяющих "большие" объемы памяти, например, при работе в режиме MTS. Проблема осложнялась еще и тем, что при обработке, требующей больших объемов памяти, эта память используется не так, как предполагает управление памятью в разделяемом пуле. Память в разделяемом пуле управляется на основе давности использования, что отлично подходит для кэширования и повторного использования данных. При выделении же больших объемов памяти фрагмент выделяется, используется и после этого он не нужен, т.е. нет смысла его кэшировать.

Серверу Oracle требовался аналог буферных пулов RECYCLE и KEEP в буферном кэше. Именно в таком качестве сейчас и выступают большой пул и разделяемый пул. Большой пул — это область памяти, управляемая по принципу пула RECYCLE, а разделяемый пул скорее похож на буферный пул KEEP: если фрагмент в нем используется часто, он кэшируется надолго.

Память в большом пуле организована по принципу "кучи" и управляется с помощью алгоритмов, аналогичных используемым функциями malloc() и free() в языке C. После освобождения фрагмента памяти он может использоваться другими процессами. В разделяемом пуле отсутствует понятие освобождения фрагмента памяти. Память выделяется, используется, а затем перестает использоваться. Через некоторое время, если эту память необходимо использовать повторно, сервер Oracle позволит изменить содержимое устаревшего фрагмента. Проблема при использовании только разделяемого пула состоит в том, что все потребности в памяти нельзя подогнать под одну мерку.

Большой пул, в частности, используется:

- сервером в режиме MTS для размещения области UGA в SGA;

- при распараллеливании выполнения операторов — для буферов сообщений, которыми обмениваются процессы для координации работы серверов;
- в ходе резервного копирования для буферизации дискового ввода/вывода утилиты **RMAN.**

Как видите, ни одну из описанных выше областей памяти нельзя помещать в буферный пул с вытеснением небольших фрагментов памяти на основе давности использования. Область **UGA,** например, не будет использоваться повторно по завершении сеанса, поэтому ее немедленно надо возвращать в пул. Кроме того, область **UGA** обычно — достаточно большая. Как было показано на примере, где изменялось значение параметра **SORT_AREA_RETAINED_SIZE,** область **UGA** может быть очень большой, и, конечно, больше, чем фрагмент в 4 Кбайт. При помещении области **UGA** в разделяемый пул она фрагментируется на части одинакового размера и, что хуже всего, выделение больших областей памяти, никогда не используемых повторно, приведет к выбрасыванию из пула фрагментов, которые могли бы повторно использоваться. В дальнейшем на перестройку этих фрагментов памяти расходуются ресурсы сервера.

То же самое справедливо и для буферов сообщений. После того как сообщение доставлено, в них уже нет необходимости. С буферами, создаваемыми в процессе резервного копирования, все еще сложнее: они большие и сразу после использования сервером Oracle должны "исчезать".

Использовать большой пул при работе в режиме MTS не обязательно, но желательно. Если сервер работает в режиме MTS в отсутствие большого пула, вся память выделяется из разделяемого пула, как это и было в версиях Oracle вплоть до 7.3. Из-за этого производительность со временем будет падать, поэтому такой конфигурации надо избегать. Большой пул стандартного размера будет создаваться при установке одного из следующих параметров инициализации: **DBWn_IO_SLAVES** или **PARALLEL_AUTOMATIC_TUNING.** Рекомендуется задавать размер большого пула явно. Однако стандартное значение не может использоваться во всех без исключения случаях.

Java-пул

Java-пул — это самый новый пул памяти в Oracle 8i. Он был добавлен в версии 8.1.5 для поддержки работы Java-машины в базе данных. Если поместить хранимую процедуру на языке Java или компонент EJB (Enterprise JavaBean) в базу данных, сервер Oracle будет использовать этот фрагмент памяти при обработке соответствующего кода. Одним из недостатков первоначальной реализации Java-пула в Oracle 8.1.5 было то, что он не отображался командой **SHOW SGA** и не был представлен строками в представлении **V$SGASTAT. В** то время это особенно сбивало с толку, поскольку параметр инициализации **JAVA_POOL_SIZE,** определяющий размер этой структуры, имел стандартное значение 20 Мбайт. Это заставляло людей гадать, почему область **SGA** занимает оперативной памяти на 20 Мбайт больше, чем следует.

Начиная с версии 8.1.6, однако, Java-пул виден в представлении **V$SGASTAT,** а также в результатах выполнения команды **SHOW SGA.** Параметр инициализации **JAVA_POOL_SIZE** используется для определения фиксированного объема памяти, отводящегося для Java-кода и данных сеансов. В Oracle 8.1.5 этот параметр мог иметь зна-

чения от 1 Мбайт до 1 Гбайт. В Oracle 8.1.6 и последующих версиях диапазон допустимых значений уже 32 Кбайт—1 Гбайт. Это противоречит документации, где по-прежнему указан устаревший минимум — 1 Мбайт.

Java-пул используется по-разному, в зависимости от режима работы сервера Oracle. В режиме выделенного сервера Java-пул включает разделяемую часть каждого Java-класса, использованного хоть в одном сеансе. Эти части только читаются (векторы выполнения, методы и т.д.) и имеют для типичных классов размер от 4 до 8 Кбайт.

Таким образом, в режиме выделенного сервера (который, как правило, и используется, если в приложениях применяются хранимые процедуры на языке Java) объем общей памяти для Java-пула весьма невелик; его можно определить исходя из количества используемых Java-классов. Учтите, что информация о состоянии сеансов при работе в режиме разделяемого сервера в области SGA не сохраняется, поскольку эти данные находятся в области UGA, а она, если вы помните, в режиме разделяемого сервера является частью области PGA.

При работе в режиме MTS Java-пул включает:

- разделяемую часть каждого Java-класса и
- часть области UGA для каждого сеанса, используемую для хранения информации о состоянии сеансов.

Оставшаяся часть области UGA выделяется как обычно — из разделяемого пула или из большого пула, если он выделен.

Поскольку общий размер Java-пула фиксирован, разработчикам приложений необходимо оценить общий объем памяти для приложения и умножить на предполагаемое количество одновременно поддерживаемых сеансов. Полученное значение будет определять общий размер Java-пула. Каждая Java-часть области UGA будет увеличиваться и уменьшаться при необходимости, но помните, что размер пула должен быть таким, чтобы части всех областей UGA могли поместиться в нем одновременно.

В режиме MTS, который обычно используется для приложений, использующих архитектуру CORBA или компоненты EJB (об этом говорилось в главе 1), может потребоваться очень большой Java-пул. Его размер будет зависеть не от количества используемых классов, а от количества одновременно работающих пользователей. Как и большой пул, размеры которого становятся очень большими в режиме MTS, Java-пул тоже может разрастаться до огромных размеров.

Итак, в этом разделе была рассмотрена структура памяти сервера Oracle. Мы начали с уровня процессов и сеансов, поговорили об областях PGA (Process Global Area — глобальная область процесса) и UGA (User Global Area — глобальная область пользователя) и разобрались в их взаимосвязи. Было показано, как режим, в котором пользователь подключается к серверу Oracle, определяет организацию памяти. Подключение к выделенному серверу предполагает использование памяти серверным процессом в большем объеме, чем подключение в режиме MTS, но работа в режиме MTS требует создания намного большей области SGA. Затем мы описали компоненты самой области SGA, выделив в ней шесть основных структур. Были описаны различия между разделяемым и большим пулом, и показано, почему большой пул необходим для "сохранения" разделяемого пула. Мы описали Java-пул и его использование в различных условиях. Был рассмотрен буферный кэш и способ деления его на меньшие, более специализированные пулы.

Теперь можно переходить к физическим процессам экземпляра Oracle.

Процессы

Осталось рассмотреть последний элемент "головоломки". Мы изучили организацию базы данных и набор образующих ее физических файлов. Разбираясь с использованием памяти сервером Oracle, рассмотрели половину экземпляра. Оставшийся компонент архитектуры — набор процессов, образующий вторую половину экземпляра. Некоторые из этих процессов, например процесс записи блоков в базу данных **(DBWn)** и процесс записи журнала **(LGWR),** уже упоминались. Здесь мы более детально рассмотрим функцию каждого процесса: что и почему они делают. В этом разделе "процесс" будет использоваться как синоним "потока" в операционных системах, где сервер Oracle реализован с помощью потоков. Так, например, если описывается процесс **DBWn,** в среде Windows ему соответствует поток **DBWn.**

В экземпляре Oracle есть три класса процессов.

- **Серверные процессы.** Они выполняют запросы клиентов. Мы уже затрагивали тему выделенных и разделяемых серверов. И те, и другие относятся к серверным процессам.
- **Фоновые процессы.** Это процессы, которые начинают выполняться при запуске экземпляра и решают различные задачи поддержки базы данных, такие как запись блоков на диск, поддержка активного журнала повторного выполнения, удаление прекративших работу процессов **и** т.д.
- **Подчиненные процессы.** Они подобны фоновым процессам, но выполняют, кроме того, действия от имени фонового или серверного процесса.

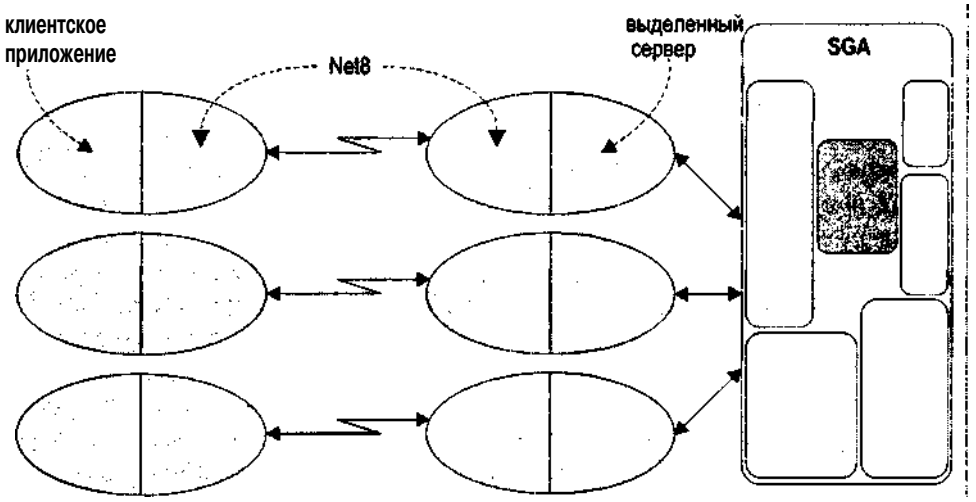
Мы рассмотрим все эти процессы и постараемся выяснить, какую роль они играют в экземпляре.

Серверные процессы

Мы уже бегло рассматривали эти процессы ранее при обсуждении выделенных и разделяемых серверов. Здесь мы еще раз опишем два вида серверных процессов и более детально рассмотрим их архитектуру.

Выделенные и разделяемые серверы решают одну и ту же задачу: обрабатывают передаваемые им SQL-операторы. При получении запроса **SELECT * FROM EMP** именно выделенный/разделяемый сервер Oracle будет разбирать его и помещать в разделяемый пул (или находить соответствующий запрос в разделяемом пуле). Именно этот процесс создает план выполнения запроса. Этот процесс реализует план запроса, находя необходимые данные в буферном кэше или считывая данные в буферный кэш с диска. Такие серверные процессы можно назвать "рабочими лошадками" СУБД. Часто именно они потребляют основную часть процессорного времени в системе, поскольку выполняют сортировку, суммирование, соединения — в общем, почти все.

В режиме выделенного сервера имеется однозначное соответствие между клиентскими сеансами и серверными процессами (или потоками). Если имеется 100 сеансов на UNIX-машине, будет 100 процессов, работающих от их имени. Графически это можно представить так:



С клиентским приложением скомпонованы библиотеки Oracle. Они обеспечивают функциональный интерфейс (Application Program Interface — API) для взаимодействия с базой данных. Функции API "знают", как передавать запрос к базе данных и обрабатывать возвращаемый курсор. Они обеспечивают преобразование запросов пользователя в передаваемые по сети пакеты, обрабатываемые выделенным сервером. Эти функции обеспечивает компонент Net8 — сетевое программное обеспечение/протокол, используемое Oracle для клиент/серверной обработки (даже в n-звенной архитектуре есть место для клиент/серверного взаимодействия). Сервер Oracle использует такую архитектуру, даже если протокол Net8 не нужен. То есть, когда клиент и сервер работают на одной и той же машине, используется эта двухпроцессорная (известная также как двухзадачная — two-task) архитектура. Эта архитектура обеспечивает два преимущества.

- **Удаленное выполнение.** Клиентское приложение, естественно, может работать не на той машине, где работает СУБД.
- **Изолирование адресных пространств.** Серверный процесс имеет доступ для чтения и записи к области SGA. Ошибочный указатель в клиентском процессе может повредить структуры данных в области SGA, если клиентский и серверный процессы физически взаимосвязаны.

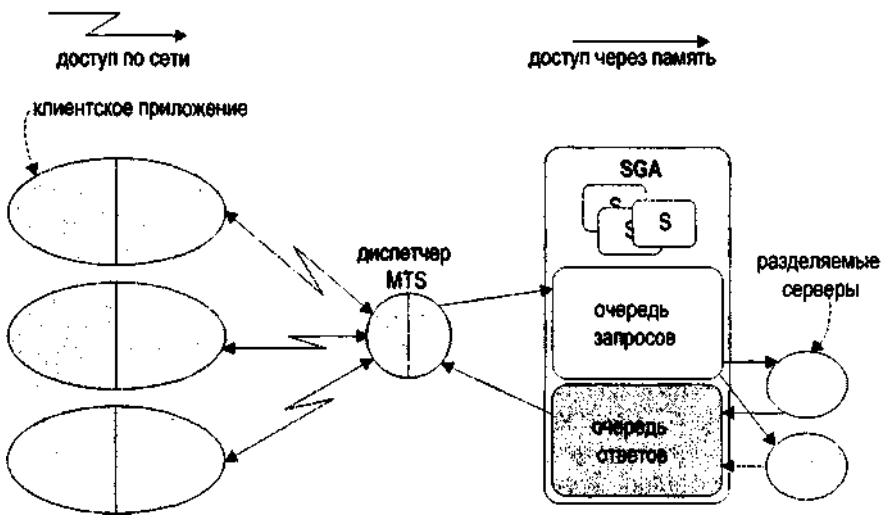
Ранее в этой главе мы рассматривали "порождение", или создание, этих серверных процессов процессом прослушивания Oracle Net8 Listener. Не будем снова возвращаться к этому процессу, но коротко рассмотрим, что происходит, если процесс прослушивания не задействован. Механизм во многом аналогичен, но вместо создания выделенного сервера процессом прослушивания с помощью вызовов **fork()/exec()** в ОС UNIX или вызова IPC (Inter Process Communication), как это происходит в Windows, процесс создается непосредственно клиентским процессом. Это можно четко увидеть в ОС UNIX:

```
ops$tkyte@ORA8I.WORLD>  select  a.spid  dedicated_server,
  2                 b.process  clientpid
  3     from v$process  a,  v$session  b
  4    where  a.addr  =  b.paddr
```

```
  5       and b.audsid = userenv('sessionid')
  6  /

DEDICATED     CLIENTPID

7055          7054

ops$tkyte@ORA8I.WORLD>  !/bin/pa  -lp  7055
 F  S   DID   PID   PPID   C  PRI  HI      ADDR      SZ     WCHAN  TTY   TIME  CMD
 8  S 30174  7055   7054   0   41  20  61ac4230  36815  639b1998  ?     0:00  oracle

ops$tkyte@ORA8I.WORLD>  !/bin/pa  -lp  7054
 F  S   UID   PID   PPID   C  PRI  N1      ADDR      SZ   WCHAN   TTY    TIME  CMD
 8  S 12997  7054   6783   0   51  20  63eece30  1087  63eece30  pts/7  0:00  sqlplus
```

Я использовал запрос для определения идентификатора процесса (PID) моего выделенного сервера (столбец **SPID** в представлении **V$PROCESS** — это идентификатор процесса операционной системы, использовавшегося для выполнения запроса). Кроме того, в столбце **PROCESS** представления **V$SESSION** находится идентификатор клиентского процесса, подключившегося к базе данных. С помощью команды ps можно явно показать, что **PPID** (Parent Process ID — идентификатор родительского процесса) моего выделенного сервера соответствует процессу SQL*Plus. В данном случае именно утилита SQL*Plus создала выделенный сервер с помощью системных вызовов **fork()** и **exec**().

Теперь давайте более детально рассмотрим другой тип серверных процессов — разделяемый серверный процесс. Для подключения к серверному процессу этого типа обязательно используется протокол Net8, даже если клиент и сервер работают на одной машине, — нельзя использовать режим MTS без процесса прослушивания Net8. Как уже описывалось ранее в этом разделе, клиентское приложение подключается к процессу прослушивания Net8 и перенаправляется на процесс-диспетчер. Диспетчер играет роль канала передачи информации между клиентским приложением и разделяемым серверным процессом. Ниже представлена схема подключения к базе данных через разделяемый сервер:



Как видите, клиентские приложения со скомпонованными в них библиотеками Oracle физически подключаются к диспетчеру MTS. Диспетчеров MTS для любого экземпляра можно сгенерировать несколько, но часто для сотен и даже тысяч пользователей используется один диспетчер. Диспетчер отвечает за получение входящих запросов от клиентских приложений и их размещение в очереди запросов в области SGA. Первый свободный разделяемый серверный процесс, по сути, ничем не отличающийся от выделенного серверного процесса, выберет запрос из очереди и подключится к области UGA соответствующего сеанса (прямоугольника с 'S' на представленной выше схеме). Разделяемый сервер обработает запрос и поместит полученный при его выполнении результат в очередь ответов. Диспетчер постоянно следит за появлением результатов в очереди и передает их клиентскому приложению. С точки зрения клиента нет никакой разницы между подключением к выделенному серверу и подключением в режиме MTS, — они работают одинаково. Различие возникает только на уровне экземпляра.

Выделенный и разделяемый сервер

Прежде чем перейти к остальным процессам, давайте обсудим, почему поддерживается два режима подключения и когда лучше использовать каждый из них. Режим выделенного сервера — наиболее широко используемый способ подключения к СУБД Oracle для всех приложений, использующих SQL-запросы. Режим выделенного сервера проще настроить и он обеспечивает самый простой способ подключения. При этом требуется минимальное конфигурирование. Настройка и конфигурирование режима MTS, хотя и несложный, но дополнительный шаг. Основное различие между этими режимами, однако, не в настройке. Оно связано с особенностями работы. При использовании выделенного сервера имеется соответствие один к одному между клиентским сеансом и серверным процессом. В режиме MTS соответствие — многие к одному (много клиентов и один разделяемый сервер). Как следует из названия, разделяемый сервер — общий ресурс, а выделенный — нет. При использовании общего ресурса необходимо стараться не монополизировать его надолго. Как было показано в главе 1, в примере с компонентами EJB, запускавшими продолжительную хранимую процедуру, монополизация этого ресурса может приводить как бы к зависанию системы. На представленной выше схеме имеется два разделяемых сервера. При наличии трех клиентов, более-менее одновременно пытающихся запустить 45-секундный процесс, два из них получат результат через 45 секунд, а третий — через 90 секунд. Правило номер один для режима MTS: убедитесь, что транзакции выполняются быстро. Они могут выполняться часто, но должны быть короткими (что обычно и бывает в системах OOT). В противном случае будут наблюдаться все признаки замедления работы системы из-за монополизации общих ресурсов несколькими процессами. В экстремальных случаях, если все разделяемые серверы заняты, система "зависает".

Поэтому режим MTS очень хорошо подходит для систем класса OOT, характеризующихся короткими, но частыми транзакциями. В системе класса OOT транзакции выполняются за миллисекунды, — ни одно действие не требует для выполнения более чем доли секунды. Режим MTS не подходит, однако, для хранилища данных. В такой системе выполняются запросы продолжительностью одна, две, пять и более минут. Для режима MTS это "смертельно". В системе, где 90 процентов задач относятся к классу OOT,

а 10 процентов — "не совсем ООТ", можно поддерживать одновременно выделенные и разделяемые серверы в одном экземпляре. В этом случае существенно сокращается количество процессов для пользователей ООТ, а "не совсем ООТ"-задачи не монополизируют надолго разделяемые серверы.

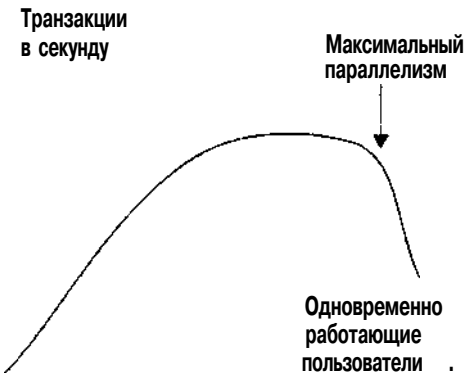
Итак, какие же преимущества дает режим MTS, если учитывать, для какого типа транзакций он предназначен? Режим MTS позволяет добиться следующего.

Сократить количество процессов/потоков операционной системы

В системе с тысячами пользователей ОС может быстро оказаться перегруженной при попытке управлять тысячами процессов. В обычной системе одновременно активна лишь небольшая часть этих тысяч пользователей. Например, я недавно работал над системой с 5000 одновременно работающих пользователей. В каждый момент времени в среднем активны были не более 50. Эта система могла бы работать с 50 разделяемыми серверными процессами, на два порядка (в 100 раз) сокращая количество процессов в операционной системе. При этом существенно сокращается количество переключений контекстов на уровне операционной системы.

Искусственно ограничить степень параллелизма

Как человеку, участвовавшему во многих тестированиях производительности, преимущества ограничения степени параллелизма мне очевидны. При тестировании клиенты просят запустить как можно больше пользователей, пока система не перестанет работать. Одним из результатов такого рода тестирования является диаграмма, показывающая зависимость количества транзакций от количества одновременно работающих пользователей:



Сначала при добавлении одновременно работающих пользователей количество транзакций растет. С какого-то момента, однако, добавление новых пользователей не увеличивает количества выполняемых в секунду транзакций: оно стабилизируется. Пропускная способность достигла максимума, и время ожидания ответа начинает расти (каждую секунду выполняется то же количество транзакций, но пользователи получают резуль-

таты со все возрастающей задержкой. При дальнейшем добавлении пользователей пропускная способность начинает падать. Количество одновременно работающих пользователей перед началом этого падения и является максимально допустимой степенью параллелизма в системе. Дальше система переполняется запросами, и образуются очереди. С этого момента система не справляется с нагрузкой. Не только существенно увеличивается время ответа, но и начинает падать пропускная способность системы. Если ограничить количество одновременно работающих пользователей до числа, непосредственно предшествующего падению, можно обеспечить максимальную пропускную способность и приемлемое время ответа для большинства пользователей. Режим MTS позволяет ограничить максимальную степень параллелизма в системе до этого количества одновременно работающих пользователей.

Сократить объем памяти, необходимый системе

Это одна из наиболее часто упоминаемых причин использования режима MTS: сокращается объем памяти, необходимой для поддержки определенного количества пользователей. Да, сокращается, но не настолько, как можно было бы ожидать. Помните, что при использовании режима MTS область **UGA** помещается в **SGA.** Это означает, что при переходе на режим MTS необходимо точно оценить суммарный объем областей **UGA** и выделить место в области **SGA** с помощью параметра инициализации **LARGE_POOL** Поэтому размер области SGA при использовании режима **MTS** обычно очень большой. Эта память выделяется заранее и поэтому может использоваться только СУБД. Сравните это с режимом разделяемого сервера, когда процессы могут использовать любую область памяти, не выделенную под SGA. Итак, если область **SGA** становится намного больше вследствие размещения в ней областей UGA, каким же образом экономится память? Экономия связана с уменьшением количества выделяемых областей **PGA.** Каждый выделенный/разделяемый сервер имеет область **PGA.** В ней хранится информация процесса. В ней располагаются области сортировки, области хешей и другие структуры процесса. Именно этой памяти для системы надо меньше, если используется режим MTS. При переходе с 5000 выделенных серверов на 100 разделяемых освобождается 4900 областей **PGA** — именно такой объем памяти и экономится в режиме MTS.

Конечно, используют в этих целях режим MTS только при отсутствии выбора. Если необходимо взаимодействовать с компонентами EJB в базе данных, придется использовать режим MTS. Есть и другие расширенные возможности подключения, требующие использования режима MTS. Если необходимо централизовать связи нескольких баз данных, например, также придется использовать режим MTS.

Рекомендация

Если система не перегружена и нет необходимости использовать режим **MTS** для обеспечения необходимой функциональности, лучше использовать выделенный сервер. Выделенный сервер проще устанавливать, и упрощается настройка производительности. Есть ряд операций, которые можно выполнять только при подключении в режиме выделенного сервера, так что в любом экземпляре надо поддерживать либо оба режима, либо только режим выделенного сервера.

С другой стороны, если необходимо поддерживать большое количество пользователей и **известно,** что эксплуатировать систему придется в режиме MTS, я рекомендую **разрабатывать** и **тестировать** ее тоже в режиме MTS. Если система разрабатывалась в режиме выделенного сервера и никогда не тестировалась в режиме MTS, вероятность неудачи повышается. Испытывайте систему в рабочих условиях; тестируйте ее производительность; проверьте, хорошо ли она работает в режиме MTS. То есть, проверьте, не монополизирует ли она надолго разделяемые серверы. Обнаруженные на стадии разработки недостатки устранить гораздо проще, чем при внедрении. Для сокращения времени работы процесса можно использовать средства расширенной обработки очередей (Advanced Queues — AQ), но это надо **учесть в проекте** приложения. Такие вещи лучше делать на этапе разработки.

Если в приложении уже используется пул подключений (например, пул подключений компонентов JEE) и размер этого пула определен верно, использование режима MTS только снизит производительность. Размер пула подключений уже рассчитан с учетом максимального количества одновременных подключений, поэтому необходимо, чтобы каждое из этих подключений выполнялось непосредственно к выделенному серверу. Иначе один пул подключений будет просто подключаться к другому пулу подключений.

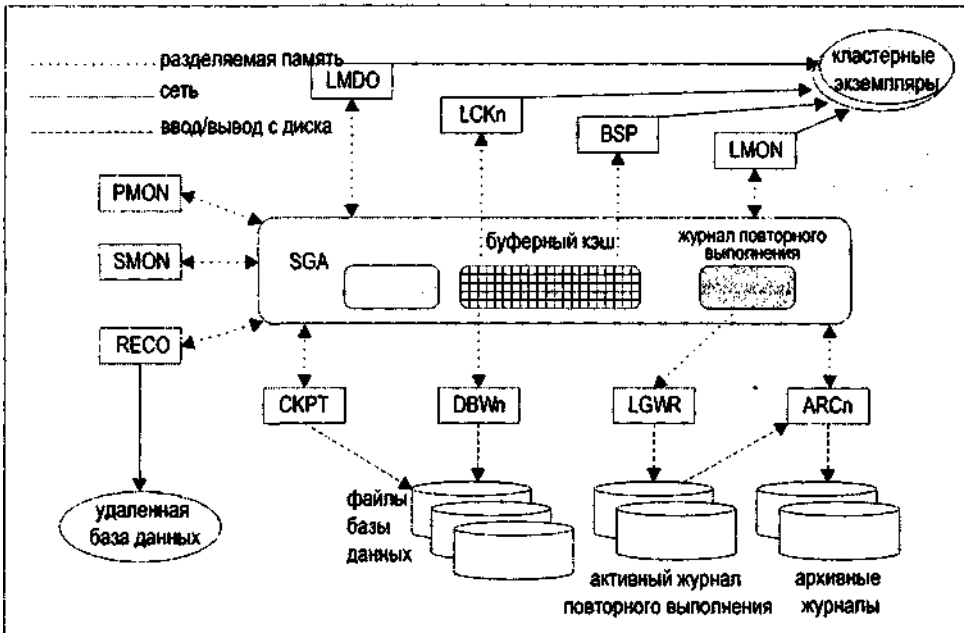
Фоновые процессы

Экземпляр Oracle состоит из двух частей: области SGA и набора фоновых процессов. Фоновые процессы выполняют рутинные задачи сопровождения, обеспечивающие работу СУБД. Есть, например, процесс, автоматически поддерживающий буферный кэш и при необходимости записывающий блоки данных на диск. Есть процесс, копирующий заполненный файл активного журнала повторного выполнения в архив. Еще один процесс отвечает за очистку всех структур, которые использовались завершившимися процессами, и т.д. Каждый из этих процессов решает конкретную задачу, но работает в координации с остальными. Например, когда процесс, записывающий файлы журнала, заполняет один журнал и переходит на следующий, он уведомляет процесс, отвечающий за архивирование заполненного журнала, что для него есть работа.

Есть два класса фоновых процессов: предназначенные исключительно для решения конкретных задач (как только что описанные) и решающие множество различных задач. Например, есть фоновый процесс, обеспечивающий работу внутренних очередей заданий в Oracle. Этот процесс контролирует очередь заданий и выполняет находящиеся в ней задания. Во многом он похож на выделенный сервер, но без подключения к клиенту. Сейчас мы рассмотрим все эти фоновые процессы, начиная с тех, которые выполняют конкретную задачу, а затем перейдем к процессам "общего назначения".

Фоновые процессы, предназначенные для решения конкретных задач

На следующей схеме представлены фоновые процессы экземпляра Oracle, имеющие конкретное назначение:



Вы не обязательно увидите все эти процессы сразу после запуска своего экземпляра, но большинство из них работает в каждом экземпляре. Процесс **ARCn** (архиватор) запускается только при работе в режиме архивирования журналов (Archive Log Mode) при включенном автоматическом архивировании. Процессы **LMD0, LCKn, LMON** и **BSP** (подробнее о них — ниже) запускаются только при работе с Oracle Parallel Server (конфигурация сервера Oracle, поддерживающая несколько экземпляров на различных машинах в кластерной среде), если открывается одна и та же база данных. Для простоты на схеме не показаны процессы диспетчеров MTS **(Dnnn)** и разделяемых серверов **(Snnn).** Поскольку мы только что детально их рассмотрели, они не показаны, чтобы упростить схему. Предыдущая схема показывает, что можно "увидеть" при запуске экземпляра Oracle, если база данных смонтирована и открыта. Например, в моей UNIX-системе сразу после запуска экземпляра имеются следующие процессы:

```
/bin/ps -aef | grep 'ora_ * ora8i$'
ora816 20642     1  0   Jan 17 ?     5:02 ora_arc0_ora8i
ora816 20636     1  0   Jan 17 ?   265:44 ora_snp0_ora8i
ora816 20628     1  0   Jan 17 ?    92:17 ora_lgwr_ora8i
ora816 20626     1  0   Jan 17 ?     9:23 ora_dbw0_ora8i
ora816 20638     1  0   Jan 17 ?     0:00 ora_s000_oraBi
ora816 20634     1  0   Jan 17 ?     0:04 ora_reco_ora8i
ora816 20630     1  0   Jan 17 ?     6:56 ora_ckpt_ora8i
ora816 20632     1  0   Jan 17 ?   186:44 ora_smon_ora8i
ora816 20640     1  0   Jan 17 ?     0:00 ora_d000_ora8i
ora816 20624     1  0   Jan 17 ?     0:05 ora_pmon_ora8i
```

Они соответствуют процессам, представленным на схеме, за исключением процесса **SNPn** (о нем будет рассказано позже, т.к. он не является фоновым процессом, выпол-

няющим "конкретную" задачу). Обратите внимание на соглашение по именованию этих процессов. Имя процесса начинается с префикса **ora_.** Затем следуют четыре символа, представляющие фактическое имя процесса, а затем суффикс — **_ora8i.** Дело в том, что у меня параметр инициализации **ORACLE_SID** (идентификатор сайта) имеет значение **ora8i.** В ОС UNIX это существенно упрощает идентификацию фоновых процессов Oracle и их привязку к определенному экземпляру (в Windows простого способа для этого нет, поскольку фоновые процессы реализованы как потоки одного большого процесса). Но что самое интересное, хотя это и не очевидно по представленным результатам, — все **эти процессы реализуются одним и тем же двоичным файлом.** Вы не найдете на диске двоичный выполняемый файл **arc0,** точно так же, как не найдете файлов **LGWR** и **DBW0.** Все эти процессы реализуются файлом **oracle** (именно этот выполняемый двоичный файл запускается). Просто при запуске процессы получают такие псевдонимы, чтобы проще было идентифицировать их назначение. Это позволяет совместно использовать большую часть объектного кода на платформе UNIX. В среде Windows это вообще не имеет значения, поскольку процессы Oracle — всего лишь потоки в одном физическом процессе, поэтому все они — один большой двоичный файл.

Давайте теперь рассмотрим функции каждого процесса.

PMON - монитор процессов

Этот процесс отвечает за очистку после нештатного прекращения подключений. Например, если выделенный сервер "падает" или, получив сигнал, прекращает работу, именно процесс **PMON** освобождает ресурсы. Процесс **PMON** откатит незафиксированные изменения, снимет блокировки и освободит ресурсы в области **SGA,** выделенные прекратившему работу процессу.

Помимо очистки после прерванных подключений, процесс **PMON** контролирует другие фоновые процессы сервера Oracle и перезапускает их при необходимости (если это возможно). Если разделяемый сервер или диспетчер сбоит (прекращает работу), процесс **PMON** запускает новый процесс (после очистки структур сбойного процесса). Процесс **PMON** следит за всеми процессами Oracle и либо перезапускает их, либо прекращает работу экземпляра, в зависимости от ситуации. Например, в случае сбоя процесса записи журнала повторного выполнения **(LGWR)** экземпляр надо перезапускать. Это серьезная ошибка и самое безопасное — немедленно прекратить работу экземпляра, предоставив исправление данных штатному процессу восстановления. Это происходит очень редко, и о случившемся надо немедленно сообщить службе поддержки Oracle.

Еще одна функция процесса **PMON** в экземпляре (версия Oracle 8i) — регистрировать экземпляр в процессе прослушивания протокола Net8. При запуске экземпляра процесс **PMON** опрашивает известный порт (если явно не указан другой), чтобы убедиться, запущен и работает ли процесс прослушивания. Известный/стандартный порт, используемый сервером Oracle, — порт 1521. А что произойдет, если процесс прослушивания запущен на другом порте? В этом случае используется тот же механизм, но адрес процесса прослушивания необходимо указать явно с помощью параметра инициализации **LOCAL_LISTENER.** Если процесс прослушивания запущен, процесс **PMON** связывается с ним и передает соответствующие параметры, например имя службы.

SMON - монитор системы

SMON — это процесс, занимающийся всем тем, от чего "отказываются" остальные процессы. Это своего рода "сборщик мусора" для базы данных. Вот некоторые из решаемых им задач.

- **Очистка временного пространства.** С появлением по-настоящему временных табличных пространств эта задача упростилась, но она не снята с повестки дня полностью. Например, при построении индекса выделяемые ему в ходе создания экстенты помечаются как временные **(TEMPORARY).** Если выполнение оператора **CREATE INDEX** прекращено досрочно по какой-либо причине, процесс **SMON** должен эти экстенты освободить. Есть и другие операции, создающие временные экстенты, за очистку которых также отвечает процесс **SMON.**
- **Восстановление после сбоев.** Процесс **SMON** после сбоя восстанавливает экземпляр при перезапуске.
- **Дефрагментация свободного пространства.** При использовании табличных пространств, управляемых по словарю, процесс **SMON** заменяет расположенные подряд свободные экстенты одним "большим" свободным экстентом. Это происходит только в табличном пространстве, управляемом по словарю и имеющем стандартную конструкцию хранения с ненулевым значением параметра **PCTINCREASE.**
- **Восстановление транзакций, затрагивающих недоступные файлы.** Эта задача аналогична той, которая возникает при запуске базы данных. Процесс **SMON** восстанавливает сбойные транзакции, пропущенные при восстановлении экземпляра после сбоя по причине недоступности файлов для восстановления. Например, файл мог быть на недоступном или на не смонтированном диске. Когда файл будет доступен, процесс SMON восстановит его.
- **Восстановление сбойного экземпляра в OPS. В** конфигурации Oracle Parallel Server, если одна из машин кластера останавливается (на ней происходит сбой), другая машина в экземпляре откроет файлы журнала повторного выполнения этой сбойной машины и восстановит все данные этой машины.
- **Очистка таблицы OBJ$. OBJ$** — низкоуровневая таблица словаря данных, содержащая записи практически для каждого объекта (таблицы, индекса, триггера, представления и т.д.) базы данных. Часто там встречаются записи, представляющие удаленные или "отсутствующие" объекты, используемые механизмом поддержки зависимостей Oracle. Процесс **SMON** удаляет эти ненужные строки.
- **Сжатие сегментов отката.** Процесс **SMON** автоматически сжимает сегмент отката до заданного размера.
- **"Отключение" сегментов отката.** Администратор базы данных может "отключить" или сделать недоступным сегмент отката с активными транзакциями. Активные транзакции могут продолжать использование такого отключенного сегмента отката. В этом случае сегмент отката фактически не отключается: он помечается для

"отложенного отключения". Процесс SMON периодически пытается "действительно" отключить его, пока это не получится.

Этот список дает представление о том, что делает процесс **SMON.** Как видно из представленной выше информации о процессах, полученной с помощью команды ps, процесс **SMON** может со временем потребовать существенных вычислительных ресурсов (команда ps выполнялась на машине, где экземпляр проработал около месяца). Процесс **SMON** периодически "пробуждается" (или его "будят" другие фоновые процессы) для выполнения задач сопровождения.

RECO - восстановление распределенной базы данных

Процесс **RECO** имеет очень конкретную задачу: он восстанавливает транзакции, оставшиеся в готовом состоянии из-за сбоя или потери связи в ходе *двухэтапной фиксации* (2PC). 2PC — это распределенный протокол, позволяющий неделимо фиксировать изменения в нескольких удаленных базах данных. Он пытается максимально снизить вероятность распределенного сбоя перед фиксацией. При использовании протокола 2PC между N базами данных одна из баз данных обычно (но не всегда) та, к которой первоначально подключился клиент, становится *координатором.* Соответствующий сервер опрашивает остальные N -1 серверов, готовы ли они фиксировать транзакцию. Фактически, этот сервер связывается с остальными N -1 серверами и просит их подготовиться к фиксации. Каждый из N -1 серверов сообщает о своем состоянии готовности как да (YES) или нет (NO). Если любой из серверов вернул NO, вся транзакция откатывается. Если все серверы вернули **YES,** координатор рассылает всем N -1 серверам сообщение о постоянной фиксации.

Если серверы ответили **YES** и подготовились к фиксации, но до получения директивы о фактической фиксации от координатора происходит сбой сети или возникает какая-то другая ошибка, транзакция становится *сомнительной* (in-doubt) распределенной транзакцией. Протокол 2PC старается сократить до минимума время, в течение которого это может произойти, но не может полностью предотвратить сомнительные транзакции. Если сбой произойдет в определенном месте и в определенное время, дальнейшую обработку сомнительной транзакции выполняет процесс **RECO.** Он пытается связаться с координатором транзакции, чтобы узнать ее исход. До этого транзакция остается незафиксированной. Связавшись с координатором транзакции, процесс **RECO** восстановит либо откатит ее.

Если связаться с координатором долго не удается и имеется ряд сомнительных транзакций, их можно зафиксировать или откатить вручную. Это приходится делать, поскольку сомнительная распределенная транзакция может вызвать блокирование читающих пишущими (единственный случай в СУБД Oracle). Ваш администратор базы данных должен связаться с администратором другой базы данных и попросить его определить состояние сомнительных транзакций. Затем администратор базы данных может зафиксировать или откатить их, предоставив все остальное процессу **RECO.**

CKPT — *обработка контрольной точки*

Процесс обработки контрольной точки вовсе не обрабатывает ее, как можно предположить по названию, — это делает процесс **DBWn.** Процесс **CKPT** просто содейству-

ет обработке контрольной точки, обновляя заголовки файлов данных. Раньше процесс **CKPT** был необязательным, но, начиная с версии 8.0, он запускается всегда, так что он представлен в результатах выполнения команды ps в ОС UNIX. Ранее заголовки файлов данных обновлялись в соответствии с информацией о контрольной точке процессом записи журнала **LGWR** (Log Writer). Однако с ростом размеров баз данных и увеличением количества файлов это стало невыполнимой задачей для процесса **LGWR.** Если процессу **LGWR** надо обновлять десятки, сотни, а то и тысячи файлов, увеличивается вероятность того, что ожидающие фиксации транзакций сеансы будут ждать слишком долго. Процесс **CKPT** снимает эту задачу с процесса LGWR.

DBWn - запись блоков базы данных

Процесс записи блоков базы данных (Database Block Writer — **DBWn)** — фоновый процесс, отвечающий за запись измененных блоков на диск. Процесс **DBWn** записывает измененные блоки из буферного кэша, чтобы освободить пространство в кэше (чтобы освободить буферы для чтения других данных) или в ходе обработки контрольной точки (чтобы перенести вперед позицию в активном файле журнала повторного выполнения, с которой сервер Oracle начнет чтение при восстановлении экземпляра после сбоя). Как было описано ранее, при переключении журнальных файлов сервером Oracle запрашивается обработка контрольной точки. Серверу Oracle нужно перенести отметку контрольной точки, чтобы не было необходимости в только что заполненном активном файле журнала повторного выполнения. Если ему не удастся это сделать до того, как возникнет необходимость в файле журнала повторного выполнения, выдается сообщение, что обработка контрольной точки не завершена (checkpoint not complete), и придется ждать завершения обработки.

Как видите, производительность процесса **DBWn** может иметь принципиальное значение. Если он недостаточно быстро записывает блоки для освобождения буферов, сеансам приходится ждать события **FREE_BUFFER_WAITS,** и показатель **'Write Complete Waits'** начинает расти.

Можно сконфигурировать несколько (до десяти) процессов **DBWn (DBW0 ... DBW9).** В большинстве систем работает только один процесс записи блоков базы данных, но в больших, многопроцессорных системах имеет смысл использовать несколько. Если сконфигурировано более одного процесса **DBWn,** не забудьте также увеличить значение параметра инициализации **DB_BLOCK_LRU_LATCHES**. Он определяет количество защелок *списков по давности использования,* LRU lists (теперь, в версии 8i, их называют *списками количества обращений* — touch lists). Каждый процесс **DBWn** должен иметь собственный список. Если несколько процессов **DBWn** совместно используют один список блоков для записи на диск, они будут конфликтовать друг с другом при доступе к списку.

Обычно процесс **DBWn** использует асинхронный ввод/вывод для записи блоков на диск. При использовании асинхронного ввода/вывода процесс **DBWn** собирает пакет блоков для записи и передает его операционной системе. Процесс **DBWn** не ждет, пока ОС запишет блоки, — он собирает следующий пакет для записи. Завершив асинхронную запись, ОС уведомляет об этом процесс **DBWn.** Это позволяет процессу **DBWn** работать намного быстрее, чем при последовательном выполнении действий. В разделе "Подчиненные процессы" будет показано, как с помощью подчиненных процессов вво-

да/вывода можно эмулировать асинхронный ввод/вывод на платформах, где он не поддерживается.

И последнее замечание о процессе DBWn. Он, по определению, записывает блоки, разбросанные по всему диску, — процесс DBWn выполняет множество записей вразброс. В случае изменений будут изменяться разбросанные блоки индекса и блоки данных, достаточно случайно распределенные по диску. Процесс LGWR, напротив, выполняет в основном запись последовательных блоков в журнал повторного выполнения. Это — важное отличие и одна из причин, почему сервер Oracle имеет журнал повторного выполнения и отдельный процесс LGWR. Записи вразброс выполняются намного медленнее, чем последовательные записи. Имея грязные блоки в буферном кэше в SGA и процесс LGWR, записывающий большое количество последовательных блоков информации для восстановления измененных буферов, можно повысить производительность. Сочетание работы двух процессов — процесс DBWn медленно работает в фоновом режиме, тогда как процесс LGWR быстро выполняет работу для ожидающего пользователя — позволяет повысить общую производительность. Это верно даже несмотря на то, что сервер Oracle может выполнять больший объем ввода/вывода, чем надо (записывает в журнал и в файл данных), — записи в активный журнал повторного выполнения можно пропустить, если в ходе обработки контрольной точки сервер Oracle уже записал измененные блоки на диск.

LGWR — запись журнала

Процесс LGWR отвечает за сброс на диск содержимого буфера журнала повторного выполнения, находящегося в области SGA. Он делает это:

- раз в три секунды;
- при фиксации транзакции;
- при заполнении буфера журнала повторного выполнения на треть или при записи в него 1 Мбайт данных.

Поэтому создание слишком большого буфера журнала повторного выполнения не имеет смысла: сервер Oracle никогда не сможет использовать его целиком. Все журналы записываются последовательно, а не вразброс, как вынужден выполнять ввод/вывод процесс DBWn. Запись большими пакетами, как в этом случае, намного эффективнее, чем запись множества отдельных блоков в разные части файла. Это одна из главных причин выделения процесса LGWR и журнала повторного выполнения. Эффективность последовательной записи измененных байтов перевешивает расход ресурсов на дополнительный ввод/вывод. Сервер Oracle мог бы записывать блоки данных непосредственно на диск при фиксации, но это потребовало бы записи множества разбросанных блоков, а это существенно медленнее, чем последовательная запись изменений процессом LGWR.

ARCn — архивирование

Задача процесса ARCn — копировать в другое место активный файл журнала повторного выполнения, когда он заполняется процессом LGWR. Эти архивные файлы журнала повторного выполнения затем можно использовать для восстановления носителя.

Тогда как активный журнал повторного выполнения используется для "исправления" файлов данных в случае сбоя питания (когда прекращается работа экземпляра), архивные журналы повторного выполнения используются для восстановления файлов данных в случае сбоя диска. Если будет потерян диск, содержащий файл данных **/d0l/oradata/ora8i/system.dbf,** можно взять резервные копии за прошлую неделю, восстановить из них старую копию файла и попросить сервер применить активный журнал повторного выполнения и все архивные журналы, сгенерированные с момента создания этой резервной копии. Это "подтянет" файл по времени к остальным файлам в базе данных, и можно будет продолжить работу без потери данных.

Процесс **ARCn** обычно копирует активный журнал повторного выполнения в несколько мест (избыточность — гарантия сохранности данных!). Это могут быть диски на локальной машине или, что лучше, на другой машине, на случай катастрофического сбоя. Во многих случаях архивные файлы журнала повторного выполнения копируются затем другим процессом на третье устройство хранения, например на ленту. Они также могут отправляться на другую машину для применения к резервной базе данных (это одно из средств защиты от сбоев, предлагаемое Oracle).

BSP — сервер блоков

Этот процесс используется исключительно в среде Oracle Parallel Server (OPS). OPS — конфигурация Oracle, при которой несколько экземпляров монтируют и открывают одну и ту же базу данных. Каждый экземпляр Oracle в этом случае работает на своей машине в кластере, и все они имеют доступ для чтения и записи к одному и тому же набору файлов базы данных.

При этом буферные кэши в SGA экземпляров должны поддерживаться в согласованном состоянии. Для этого и предназначен процесс BSP. В ранних версиях OPS согласование достигалось с помощью сброса блока ('ping'). Если машине в кластере требовалось согласованное по чтению представление блока данных, заблокированного в исключительном режиме другой машиной, выполнялся обмен данными с помощью сброса на диск. В результате получалась очень дорогостоящая операция чтения данных. Сейчас, при наличии процесса BSP, обмен происходит из кэша в кэш через высокоскоростное соединение машин в кластере.

LMON — контроль блокировок

Этот процесс используется исключительно в среде OPS. Процесс **LMON** контролирует все экземпляры кластера для выявления сбоя экземпляра. Затем он вместе с диспетчером распределенных блокировок (Distributed Lock Manager — DLM), используемым аппаратным обеспечением кластера, восстанавливает глобальные блокировки, которые удерживаются сбойным экземпляром.

LMD - демон диспетчера блокировок

Этот процесс используется исключительно в среде OPS. Процесс **LMD** управляет глобальными блокировками и глобальными ресурсами для буферного кэша в кластерной среде. Другие экземпляры посылают локальному процессу **LMD** запросы с требованием снять блокировку или определить, кто ее установил. Процесс **LMD** также выявляет и снимает глобальные взаимные блокировки.

LCKn - блокирование

Процесс **LCKn** используется исключительно в среде OPS. Он подобен по функциям описанному выше процессу **LMD,** но обрабатывает запросы ко всем остальным глобальным ресурсам, кроме буферного кэша.

Служебные фоновые процессы

Эти фоновые процессы необязательны — они запускаются в случае необходимости. Они реализуют средства, необязательные для штатного функционирования базы данных. Использование этих средств инициируется явно или косвенно, при использовании возможности, требующей их запуска.

Служебных фоновых процессов — два. Один из них запускает посланные на выполнение задания. В СУБД Oracle встроена очередь пакетных заданий, позволяющая выполнять по расписанию однократные или периодические задания. Другой процесс поддерживает и обрабатывает таблицы очереди, используемые средствами *расширенной поддержки очередей* (Advanced Queuing — AQ). Средства AQ обеспечивают встроенные возможности обмена сообщениями между сеансами базы данных.

Эти процессы можно увидеть в среде ОС UNIX, как и любой другой фоновый процесс, с помощью команды ps. В представленных ранее результатах выполнения команды ps можно видеть, что у меня в экземпляре работает один процесс очереди заданий **(ora_snp0_ora8i)** и ни одного процесса очереди.

SNPn - обработка снимков (очереди заданий)

Сейчас можно сказать, что имя для процесса **SNPn** выбрано неудачно. В версии 7.0 сервера Oracle впервые поддерживалась репликацию. Это делалось с помощью объекта базы данных, известного как *моментальный снимок* (snapshot). Внутренним механизмом для обновления или приведения к текущему состоянию моментальных снимков был **SNPn** — *процесс обработки снимков* (snapshot process). Этот процесс контролировал таблицу заданий, по которой определял, когда необходимо обновлять моментальные снимки в системе. В Oracle 7.1 корпорация Oracle открыла это средство для общего доступа через пакет **DBMS_JOB.** То, что было связано с моментальными снимками в версии 7.0, стало "очередью заданий" в версии 7.1 и последующих. Со временем имена параметров для управления очередью (как часто ее надо проверять и сколько процессов может быть в очереди) изменились со **SNAPSHOT_REFRESH_INTERVAL** и **SNAPSHOT_REFRESH_PROCESSES** на **JOB_QUEUE_INTERVAL** и **JOB_QUEUE_PROCESSES.** А вот имя процесса операционной системы не изменилось.

Можно иметь до 36 процессов очереди заданий. Они именуются **SNP0, SNP1,** ..., SNP9, SNPA, ..., SNPZ. Эти процессы очередей заданий интенсивно используются при репликации в ходе обновления моментального снимка или материализованного представления. Разработчики также часто используют их для запуска отдельных (фоновых) или периодически выполняющихся заданий. Например, далее в книге будет показано, как использовать очереди заданий для существенного ускорения обработки: за счет дополнительной работы в одном месте можно сделать намного приятнее среду для пользователя (аналогично тому, как сделано в самом сервере Oracle при использовании процессов **LGWR** и **DBWn).**

Процессы **SNPn** сочетают в себе особенности как разделяемого, так и выделенного сервера: обрабатывают несколько заданий, но памятью управляют как выделенный сервер (область **UGA** находится в области **PGA** процесса). Процесс очереди заданий выполняет в каждый момент времени только одно задание. Вот почему необходимо несколько процессов, если требуется выполнять несколько заданий одновременно. На уровне заданий не поддерживаются потоки или вытеснение. Запущенное задание выполняется, пока не будет выполнено (или не произойдет сбой). В приложении А мы более детально рассмотрим пакет **DBMS_JOB** и нетрадиционное использование очереди заданий.

QMNn — монитор очередей

Процесс **QMNn** по отношению к таблицам AQ выполняет ту же роль, что и процесс **SNPn** по отношению к таблице заданий. Этот процесс контролирует очереди и уведомляет ожидающие сообщений процессы о том, что доступно сообщение. Он также отвечает за распространение очередей — возможность переместить сообщение, поставленное в очередь в одной базе данных, в другую базу данных для извлечения из очереди.

Монитор очередей — это необязательный фоновый процесс. Параметр инициализации **AQ_TM_PROCESS** позволяет создать до десяти таких процессов с именами **QMN0,**..., **QMN9.** По умолчанию процессы **QMNn** не запускаются.

EMNn — монитор событий

Процессы **EMNn** — часть подсистемы расширенной поддержки очередей. Они используются для уведомления подписчиков очереди о сообщениях, в которых они могут быть заинтересованы. Это уведомление выполняется асинхронно. Имеются функции Oracle Call Interface (OCI) для регистрации обратного вызова, уведомляющего о сообщении. *Обратный вызов* — это функция в программе OCI, которая вызывается автоматически при появлении в очереди определенного сообщения. Фоновый процесс **EMNn** используется для уведомления подписчика. Процесс EMNn запускается автоматически при выдаче первого уведомления в экземпляре. После этого приложение может явно вызвать **message_receive(dequeue)** для извлечения сообщения из очереди.

Подчиненные процессы

Теперь мы готовы рассмотреть последний класс процессов Oracle — подчиненные процессы. В сервере Oracle есть два типа подчиненных процессов — ввода/вывода (I/O slaves) и параллельных запросов (Parallel Query slaves).

Подчиненные процессы ввода/вывода

Подчиненные процессы ввода/вывода используются для эмуляции асинхронного ввода/вывода в системах или на устройствах, которые его не поддерживают. Например, ленточные устройства (чрезвычайно медленно работающие) не поддерживают асинхронный ввод/вывод. Используя подчиненные процессы ввода/вывода, можно сымитировать для ленточных устройств такой способ работы, который операционная система обычно обеспечивает для дисков. Как и в случае действительно асинхронного ввода/вывода,

процесс, записывающий на устройство, накапливает большой объем данных в виде пакета и отправляет их на запись. Об их успешной записи процесс (на этот раз — подчиненный процесс ввода/вывода, а не ОС) сигнализирует исходному вызвавшему процессу, который удаляет этот пакет из списка данных, ожидающих записи. Таким образом, можно существенно повысить производительность, поскольку именно подчиненные процессы ввода/вывода ожидают завершения работы медленно работающего устройства, а вызвавший их процесс продолжает выполнять другие важные действия, собирая данные для следующей операции записи.

Подчиненные процессы ввода/вывода используются в нескольких компонентах Oracle 8i — процессы DBWn и LGWR используют их для имитации асинхронного ввода/вывода, а утилита **RMAN** (**R**ecovery **MAN**ager — диспетчер восстановления) использует их при записи на ленту.

Использование подчиненных процессов ввода/вывода управляется двумя параметрами инициализации.

- BACKUP_TAPE_IO_SLAVES. Этот параметр указывает, используются ли подчиненные процессы вода/вывода утилитой RMAN для резервного копирования или восстановления данных с ленты. Поскольку этот параметр предназначен для *ленточных* устройств, а к ленточным устройствам в каждый момент времени может обращаться только один процесс, он — булева типа, а не задает количество используемых подчиненных процессов, как можно было ожидать. Утилита RMAN запускает необходимое количество подчиненных процессов, в соответствии с количеством используемых физических устройств. Если параметр BACKUP_TAPE_IO_SLAVES имеет значение TRUE, то для записи или чтения с ленточного устройства используется подчиненный процесс ввода/вывода. Если этот параметр имеет (стандартное) значение FALSE, подчиненные процессы ввода/вывода не используются при резервном копировании. К ленточному устройству тогда обращается фоновый процесс, выполняющий резервное копирование.

- DBWn_IO_SLAVES. Задает количество подчиненных процессов ввода/вывода, используемых процессом DBWn. Процесс DBWn и его подчиненные процессы всегда записывают на диск измененные буфера буферного кэша. По умолчанию этот параметр имеет значение 0, и подчиненные процессы ввода/вывода не используются.

Подчиненные процессы параллельных запросов

В Oracle 7.1 появились средства распараллеливания запросов к базе данных. Речь идет о возможности создавать для SQL-операторов типа SELECT, CREATE TABLE, CREATE INDEX, UPDATE и т.д. план выполнения, состоящий из нескольких планов, которые можно выполнять одновременно. Результаты выполнения этих планов объединяются. Это позволяет выполнить операцию за меньшее время, чем при последовательном выполнении. Например, если имеется большая таблица, разбросанная по десяти различным файлам данных, 16-процессорный сервер, и необходимо выполнить к этой таблице запрос, имеет смысл разбить план выполнения этого запроса на 16 небольших частей и полностью использовать возможности сервера. Это принципиально отличается от использования одного процесса для последовательного чтения и обработки всех данных.

Резюме

Вот и все компоненты СУБД Oracle. Мы рассмотрели файлы, используемые в СУБД Oracle: небольшой, но важный файл параметров инициализации init.ora, файлы данных, файлы журнала повторного выполнения и т.д. Мы изучили структуры памяти, используемые экземпляром Oracle как в серверных процессах, так и в области SGA. Было показано, как различные конфигурации сервера, например подключение в режиме MTS и к выделенному серверу, принципиально влияют на использование памяти в системе. Наконец, мы рассмотрели процессы (или потоки — в зависимости от базовой ОС), обеспечивающие выполнение функций сервера Oracle. Теперь мы готовы к рассмотрению других возможностей сервера Oracle — управления блокированием и одновременным доступом, и поддержки транзакций.

3

Блокирование и одновременный доступ

Одна из основных проблем при разработке многопользовательских приложений баз. данных — обеспечить одновременный доступ максимальному количеству пользователей при согласованном чтении и изменении данных каждым из них. Механизмы *блокирования* и управления *одновременным доступом,* позволяющие решить эту проблему, являются ключевыми в любой базе данных, и в СУБД Oracle они весьма эффективны. Однако реализация этих механизмов в Oracle уникальна, и разработчик приложений должен обеспечить их корректное использование в программе при манипулировании данными. В противном случае приложение будет работать не так, как предполагалось, и целостность данных может быть нарушена (как было показано в главе 1).

В этой главе мы подробно рассмотрим, как сервер Oracle блокирует данные, а также последствия выбора модели блокирования, которые необходимо учитывать при разработке многопользовательских приложений. Мы изучим уровни блокирования данных в Oracle, реализацию многовариантной согласованности по чтению и ее последствия для разработчиков приложений. Я буду сравнивать модель блокирования Oracle с другими популярными вариантами реализации — в основном для того, чтобы развеять миф, будто "блокирование на уровне строк требует дополнительных затрат ресурсов". Так происходит только в том случае, если эти затраты ресурсов связаны с соответствующей реализацией.

Что такое блокировки?

Блокировка — это механизм, используемый для управления одновременным доступом к общему ресурсу. Обратите внимание: использован термин "общий ресурс", а не "строка таблицы". Сервер Oracle действительно блокирует данные таблицы на уровне строк, но для обеспечения одновременного доступа к различным ресурсам он использует блокировки и на других уровнях. Например, при выполнении хранимой процедуры она блокируется в режиме, который позволяет другим сеансам ее выполнять, запрещая при этом изменять ее. Блокировки используются в базе данных для одновременного доступа к общим ресурсам и обеспечения при этом целостности и согласованности данных.

В однопользовательской базе данных блокировки не нужны. У них по определению только один пользователь, изменяющий информацию. Однако если данные или структуры данных читаются и изменяются несколькими пользователями, важно иметь штатный механизм предотвращения одновременных изменений одного и того же фрагмента информации. Именно для этого и используется блокирование.

Очень важно понять, что способов блокирования в базе данных столько же, сколько и СУБД. Богатый опыт работы с моделью блокирования конкретной реляционной СУБД еще не означает, что вы знаете о блокировании все. Я, например, прежде чем начать работать с Oracle, использовал другие СУБД, в частности Sybase и Informix. В упомянутых СУБД для управления одновременным доступом применялись механизмы блокирования, но между их реализацией в каждой из СУБД имеются глубокие и фундаментальные различия. Чтобы продемонстрировать это, я кратко опишу свой путь от разработчика приложений Sybase до пользователя Informix и, наконец, разработчика приложений Oracle. Это давняя история, и поклонники Sybase скажут, что теперь в этой СУБД реализовано блокирование на уровне строк, но способ его реализации **коренным образом отличается** от используемого в Oracle. Сравнивать их — все равно, что сравнивать яблоки и апельсины, — отличия принципиальны.

Программируя для Sybase, я обычно не предусматривал возможность одновременной вставки данных в таблицу несколькими пользователями, поскольку в этой СУБД подобное происходило нечасто. В то время Sybase предлагала только блокирование на уровне страниц и, поскольку данные в некластеризованных таблицах обычно вставляются в последнюю страницу, одновременной вставки данных двумя пользователями никогда не происходило. Та же проблема возникает и при попытке одновременного обновления данных (поскольку **UPDATE** — это фактически **DELETE,** за которым следует INSERT). Возможно, именно поэтому СУБД Sybase по умолчанию фиксировала или откатывала изменения сразу после выполнения каждого оператора.

Кроме того, что в большинстве случаев пользователи не могли одновременно изменять ту же таблицу, при изменении таблицы заблокированными оказывались и многие запросы к таблице. Если при выполнении запроса к таблице требовалась страница, заблокированная в ходе обновления, мне приходилось ждать (иногда очень долго). Механизм блокирования был настолько неудачным, что поддержка транзакций, продолжительней тысячной доли секунды, практически отсутствовала, и при попытке их выполнения казалось, что база данных зависла. Так я приобрел множество плохих привычек.

Я понял, что транзакции — это "плохо", что фиксировать изменения надо быстро и никогда не удерживать блокировки данных. Одновременность работы достигалась ценой согласованности. Нужно было выбирать: правильный результат или быстро полученный результат. Я пришел к выводу, что достичь одновременно и того, и другого невозможно.

Когда я стал работать с СУБД Informix, дела пошли лучше, но не намного. Если я не забывал создавать таблицу с блокированием на уровне строк, мне удавалось обеспечить одновременную вставку данных в эту таблицу двумя пользователями. К сожалению, одновременная работа давалась дорого. Блокировки строк в Informix были дорогостоящими, требуя дополнительного времени и памяти. Требовалось определенное время для установки и снятия блокировок, и каждая блокировка требовала места в оперативной памяти. Кроме того, общее количество доступных системе блокировок необходимо было рассчитать до запуска СУБД. Если этого количества оказывалось достаточно, вам, можно сказать, повезло. В результате таблицы обычно создавались с блокированием на уровне страниц и, как в случае с Sybase, любая блокировка строки или страницы останавливала запросы, которым были необходимы соответствующие данные. Это приучило меня фиксировать изменения как можно быстрее. Плохие привычки, приобретенные в процессе работы с СУБД Sybase, снова пригодились, и, более того, я начал рассматривать блокировки как очень дорогой ресурс, который надо экономить. Я понял, что иногда необходимо повышать уровень блокирования вручную со строчного до табличного, чтобы избежать излишнего количества блокировок и останова системы.

Начав использовать СУБД Oracle, я и не пытался читать руководства, чтобы понять, как в ней работает механизм блокирования. Используя СУБД достаточно долго и считаясь своего рода экспертом в этой области (кроме Sybase и Informix, я работал с СУБД Ingress, DB2, Gupta SQLBase и с множеством других баз данных), я стал жертвой собственной самоуверенности. Казалось, что, поскольку я знаю, как все **должно** работать, именно так оно и будет работать. Я очень сильно ошибался.

Насколько сильно я ошибался, стало понятно в ходе тестирования производительности. На раннем этапе развития упомянутых СУБД среди их производителей было принято организовывать "сравнительные тесты" для крупных проектов, чтобы клиент понял, какая СУБД позволит справиться с задачей быстрее, более простым способом и предоставит больше возможностей. Было устроено сравнение СУБД Informix, Sybase и Oracle. СУБД Oracle тестировалась первой. Технические специалисты компании прибыли к нам, прочитали спецификации теста и начали настройку среды. Первое, что я заметил, — это их намерение использовать для записи времени выполнения тестовых задач таблицу базы данных, хотя предполагалось несколько десятков сеансов, в каждом из которых требовалось часто вставлять и изменять данные в этой таблице. Более того, они собирались еще и читать данные из этой таблицы в ходе выполнения теста! Отозвав одного из специалистов в сторонку, я спросил, не сошли ли они все с ума: зачем намеренно вносить еще одну потенциальную точку конфликтов в систему? Не выстроятся ли все процессы, участвующие в тесте, в очередь при работе с этой таблицей? Не исказим ли мы существенно результаты теста, пытаясь читать из этой таблицы в то время как другие процессы интенсивно вносят в нее изменения? Зачем вы хотите добавить все эти дополнительные блокировки, которыми системе придется управлять? У меня были десятки вопросов в стиле "а зачем вообще так делать". До тех пор пока я не подключил-

ся к Sybase или Informix и не показал, что происходит, когда два сеанса пытаются вставить данные в одну и ту же таблицу или кто-то пытается выполнить запрос к таблице, в которую другие сеансы вставляют строки (запрос возвращал ноль строк в секунду), специалистам Oracle казалось, что я несколько преувеличиваю проблемы. Разница между тем, как это делается в Oracle и в других СУБД, — феноменальна. Не стоит и говорить, что ни Informix, ни Sybase не пытались регистрировать результаты в базе данных в ходе тестирования. Они предпочли записывать результаты в обычные файлы операционной системы.

Из этой истории можно сделать два вывода: **все СУБД существенно различаются,** и при разработке приложения **необходимо подходить к каждой СУБД так, будто она для вас — первая.** То, что принято делать в одной СУБД, может оказаться ненужным или просто не работать в другой.

Работая с Oracle, вы поймете, что:

- Транзакции — это хорошо, именно для их поддержки и создавались СУБД.
- Откладывать фиксацию транзакции можно настолько, насколько необходимо. Не надо стремиться к коротким транзакциям с целью снижения нагрузки на систему, поскольку длинные или большие по объему изменений транзакции не нагружают систему. Правило следующее: **фиксируйте транзакцию тогда, когда это необходимо, и не раньше.** Размер транзакций диктуется только бизнес-логикой.
- Удерживать блокировки данных можно столько, сколько необходимо. Для вас это — средство, а не проблема, которой надо избегать. Блокировки не являются ограниченным ресурсом.
- Блокирование на уровне строк в Oracle не приводит к дополнительным расходам ресурсов.
- Никогда не нужно повышать уровень блокировки (например, блокировать таблицу вместо блокирования строк), поскольку "так лучше для системы". В Oracle для системы так лучше не будет: ресурсы при этом не экономятся.
- Одновременный доступ к данным и их согласованность не противоречат друг другу. Всегда можно получить результаты быстро, и при том корректные.

В последующих разделах главы эти утверждения будут рассмотрены более подробно.

Проблемы блокирования

Прежде чем описывать различные типы блокировок, используемые СУБД Oracle, имеет смысл разобраться с рядом проблем блокирования, многие из которых возникают из-за неправильно спроектированных приложений, некорректно использующих (или вообще не использующих) механизмы блокирования базы данных.

Потерянные изменения

Потерянное изменение — классическая проблема баз данных. Если коротко, потерянное изменение возникает, когда происходят следующие события (в указанном порядке).

1. Пользователь 1 выбирает (запрашивает) строку данных.
2. Пользователь 2 выбирает ту же строку.
3. Пользователь 1 изменяет строку, обновляет базу данных и фиксирует изменение.
4. Пользователь 2 изменяет строку, обновляет базу данных и фиксирует изменение.

Это называется потерянным изменением, поскольку все сделанные на шаге 3 изменения будут потеряны. Рассмотрим, например, окно редактирования информации о сотруднике, позволяющее изменить адрес, номер рабочего телефона и т.д. Само приложение — очень простое: небольшое окно поиска со списком сотрудников и возможность получить детальную информацию о каждом сотруднике. Проще некуда. Так что пишем приложение, не выполняющее никакого блокирования, — только простые операторы SELECT и UPDATE.

Итак, пользователь (пользователь 1) переходит к окну редактирования, изменяет там адрес, щелкает на кнопке Save и получает подтверждение успешного обновления. Все отлично, кроме того, что, проверяя на следующий день эту запись, чтобы послать сотруднику налоговую декларацию, пользователь 1 увидит в ней старый адрес. Как это могло случиться? К сожалению, очень просто: другой пользователь (пользователь 2) запросил ту же запись за 5 минут до того, как к ней обратился пользователь 1, и у него на экране отображались старые данные. Пользователь 1 запросил данные, изменил их, получил подтверждение изменения и даже выполнил повторный запрос, чтобы увидеть эти изменения. Однако затем пользователь 2 изменил поле номера рабочего телефона и щелкнул на кнопке сохранения, не зная, что переписал старые данные поверх внесенных пользователем 1 изменений адреса! Так может случиться потому, что разработчик приложения предпочел обновлять сразу все столбцы, а не разбираться, какой именно столбец был изменен, и написал программу так, что при изменении одного из полей обновляются все.

Обратите внимание, что для потери изменений пользователям 1 и 2 вовсе не обязательно работать с записью одновременно. Нужно, чтобы они работали с ней **примерно** в одно и то же время.

Эта проблема баз данных проявляется постоянно, когда разработчики графических интерфейсов, не имеющие достаточного опыта работы с базами данных, получают задание создать приложение для базы данных. Они получают общее представление об операторах SELECT, INSERT, UPDATE и DELETE и начинают писать программы. Когда получившееся в результате приложение ведет себя, как описано выше, пользователи полностью перестают ему доверять, особенно потому что подобные результаты кажутся случайными и спорадическими и абсолютно невоспроизводимы в управляемой среде тестирования (что приводит разработчика к мысли, что это, возможно, ошибка пользователя).

Многие инструментальные средства, например Oracle Forms, автоматически защищают разработчиков от таких ситуаций, проверяя, не изменилась ли запись с момента запроса и заблокирована ли перед началом изменений. Но другие средства разработки (и обычные программы на языке VB или Java) такой защиты не обеспечивают. Как средства разработки, обеспечивающие защиту (за кадром), так и разработчики, использую-

щие другие средства (явно), должны применять один из двух описанных ниже методов блокирования.

Пессимистическое блокирование

Этот метод блокирования должен использоваться непосредственно перед изменением значения на экране, например, когда пользователь выбирает определенную строку с целью изменения (допустим, щелкая на кнопке в окне). Итак, пользователь запрашивает данные без блокирования:

```
scott@TKYTE816> SELECT EMPNO, ENAME, SAL FROM EMP WHERE DEPTNO = 10;

    EMPNO ENAME         SAL

     7782 CLARK        2450
     7839 KING         5000
     7934 MILLER       1300
```

В какой-то момент пользователь выбирает строку для потенциального изменения. Пусть в этом случае он выбрал строку, соответствующую сотруднику MILLER. Наше приложение в этот момент (перед выполнением изменений на экране) выполняет следующую команду:

```
scott@TKYTE816> SELECT EMPNO, ENAME, SAL
  2     FROM  EMP
  3    WHERE  EMPNO =  :EMPNO
  4      AND  ENAME =  :ENAME
  5      AND  SAL =  :SAL
  6   FOR   UPDATE   NOWAIT

  7   /

      EMPNO  ENAME         SAL

       7934  MILLER       1300
```

Приложение передает значения для связываемых переменных в соответствии с данными на экране (в нашем случае — 7934, MILLER и 1300) и повторно запрашивает ту же самую строку из базы данных, но в этот раз блокирует ее изменения другими сеансами. Вот почему такой подход называется *пессимистическим* блокированием. Мы блокируем строку перед попыткой изменения, поскольку сомневается, что она останется неизменной.

Поскольку все таблицы имеют первичный ключ (приведенный выше оператор SELECT выберет не более одной строки, поскольку критерий выбора включает первичный ключ EMPNO), а первичные ключи должны быть неизменны, при выполнении этого оператора возможен один из трех результатов.

- Если данные не изменились, мы получим ту же строку сотрудника MILLER, на этот раз заблокированную от изменения (но не чтения) другими сеансами.
- Если другой сеанс находится в процессе изменения данной строки, мы получим сообщение об ошибке ORA-00054 Resource Busy (ресурс занят). Наш сеанс заб-

локирован и мы должны ждать, пока другой сеанс не завершит изменения строки.

- Если за период между выборкой данных и попыткой их изменить другой сеанс уже изменил соответствующую строку, мы получим в результате ноль строк. Данные на экране не обновятся. Приложение должно **повторно запросить** и заблокировать данные, прежде чем разрешить пользователю изменять их, чтобы предотвратить описанную выше ситуацию с потерей изменений. В этом случае, если используется пессимистическое блокирование, когда пользователь 2 пытается изменить поле номера телефона, приложение "поймет", что изменилось поле адреса, и повторно запросит данные. Поэтому пользователь 2 никогда не перезапишет старые данные поверх изменений, внесенных в это поле пользователем 1.

После успешного блокирования строки приложение выполняет требуемые изменения и фиксирует их:

```
scott@TKYTE816> UPDATE EMP
  2  SET ENAME = :ENAME, SAL = :SAL
  3  WHERE EMPNO = .EMPNO;
1 row updated.

scott@TKYTE816> commit;
Commit complete.
```

Теперь мы абсолютно безопасно изменили соответствующую строку. Мы не могли стереть изменения, сделанные в другом сеансе, поскольку проверили, что данные не изменились с момента первоначального чтения до момента блокирования.

Оптимистическое блокирование

Второй метод, который называют *оптимистическим* блокированием, состоит в том, чтобы сохранять старое и новое значения в приложении и использовать их при изменении следующим образом:

```
Update table
   Set column1 = :new_column1, column2 = :new_column2, ....
 Where column1 = :old_column1
   And column2 = :old_column2
```

Здесь мы оптимистически надеемся, что данные не изменились. Если в результате изменена **одна** строка, значит, нам повезло: данные не изменились с момента считывания. Если изменено **ноль** строк, мы проиграли — кто-то уже изменил данные и необходимо решить, что делать, чтобы это изменение не потерять. Должны ли мы заставлять пользователя повторно выполнять транзакцию после запроса новых значений для строки (сбивая его с толку, поскольку снова может получиться так, что изменяемая им строка обновлена другим сеансом)? Стоит ли попытаться совместить изменения, разрешая конфликты двух изменений на основе бизнес-правил (что требует написания большого объема кода)? Конечно, для отключившихся пользователей последний вариант — единственно возможный.

Стоит заметить, что и в этом случае тоже можно использовать оператор **SELECT FOR UPDATE NOWAIT.** Представленный выше оператор **UPDATE** позволяет избежать потери изменений, но может приводить к блокированию, "зависая" в ожидании завершения изменения строки другим сеансом. Если все приложения используют оптимистическое блокирование, то применение простых операторов UPDATE вполне допустимо, поскольку строки блокируются на очень короткое время выполнения и фиксации изменений. Однако если некоторые приложения используют пессимистическое блокирование, удерживая блокировки строк достаточно долго, имеет смысл выполнять оператор **SELECT FOR UPDATE NOWAIT** непосредственно перед оператором **UPDATE,** чтобы избежать блокирования другим сеансом.

Итак, какой же метод лучше? По моему опыту, пессимистическое блокирование очень хорошо работает в Oracle (но вряд ли так же хорошо подходит для других СУБД) и имеет много преимуществ по сравнению с оптимистическим.

При использовании пессимистического блокирования пользователь может быть уверен, что изменяемые им на экране данные сейчас ему "принадлежат" — он получил запись в свое распоряжение, и никто другой не может ее изменять. Можно возразить, что, блокируя строку до изменения, вы лишаете к ней доступа других пользователей и, тем самым, существенно снижаете масштабируемость приложения. Но обновлять строку в каждый момент времени сможет только один пользователь (если мы не хотим потерять изменения). Если сначала заблокировать строку, а затем изменять ее, пользователю будет удобнее работать. Если же пытаться изменить, не заблокировав заранее, пользователь может напрасно потерять время и силы на изменения, чтобы в конечном итоге получить сообщение: "Извините, данные изменились, попробуйте еще раз". Чтобы ограничить время блокирования строки перед изменением, можно снимать блокировку в приложении, если пользователь занялся чем-то другим и некоторое время не использует строку, или использовать *профили ресурсов* (Resource Profiles) в базе данных для отключения простаивающих сеансов.

Более того, блокирование строки в Oracle не мешает ее читать, как в других СУБД; блокирование строки не мешает обычной работе с базой данных. Все это исключительно благодаря соответствующей реализации механизмов одновременного доступа и блокирования в Oracle. В других СУБД верно как раз обратное. Если попытаться использовать в них пессимистическое блокирование, ни одно приложение не будет работать. Тот факт, что в этих СУБД блокирование строки не дает возможности выполнять к ней запросы, не позволяет даже рассматривать подобный подход. Поэтому иногда приходится "забывать" правила, выработанные в процессе работе с одной СУБД, чтобы успешно разрабатывать приложения для другой.

Блокирование

Блокирование происходит, когда один сеанс удерживает ресурс, запрашиваемый другим сеансом. В результате запрашивающий сеанс будет заблокирован — он "повиснет" до тех пор, пока удерживающий сеанс не завершит работу с ресурсом. Блокирования практически всегда можно избежать. Если оказывается, что интерактивное приложение заблокировано, проблема, скорее всего, связана с ошибкой, подобной описанному выше потерянному изменению (логика работы приложения ошибочна, что и приводит к блокированию).

Блокирование в базе данных выполняют четыре основных оператора ЯМД: INSERT, UPDATE, DELETE и SELECT FOR UPDATE. Решение проблемы в последнем случае тривиально: добавьте конструкцию **NOWAIT,** и оператор SELECT **FOR** UPDATE больше не будет заблокирован. Вместо этого приложение должно сообщать пользователю, что строка уже заблокирована. Интерес представляют остальные три оператора ЯМД. Мы рассмотрим каждый из них и увидим, почему они не должны блокировать друг друга и как это исправить, если блокирование все-таки происходит.

Заблокированныевставки

Единственный случай блокирования операторами INSERT друг друга, — когда имеется таблица с первичным ключом или ограничением уникальности и два сеанса одновременно пытаются вставить строку с одним и тем же значением. Один из сеансов будет заблокирован, пока другой не зафиксирует изменение (в этом случае заблокированный сеанс получит сообщение об ошибке, связанной с дублированием значения) или не откатит его (в этом случае операция заблокированного сеанса будет выполнена успешно). Такое обычно происходит с приложениями, позволяющими генерировать пользователю первичные ключи или значения уникальных столбцов. Этой проблемы проще всего избежать за счет использования при генерации первичных ключей последовательностей Oracle — средства генерации уникальных ключей, обеспечивающего максимальный параллелизм в многопользовательской среде. Если нельзя использовать последовательность, можно применить метод, описанный в приложении А при рассмотрении пакета **DBMS_LOCK.** Там я демонстрирую, как решить эту проблему, прибегнув к явному блокированию вручную.

Заблокированные изменения и удаления

В интерактивном приложении, которое запрашивает данные из базы, позволяет пользователю манипулировать ими, а затем возвращает их в базу данных, заблокированные операторы **UPDATE** или **DELETE** показывают, что в коде может быть проблема потерянного изменения. Вы пытаетесь изменить с помощью UPDATE строку, которую уже изменяет другой пользователь, другими словами, которая уже кем-то заблокирована. Этого блокирования можно избежать с помощью запроса SELECT **FOR** UPDATE NOWAIT, позволяющего:

- проверить, не изменились ли данные с момента их прочтения (для предотвращения потерянного изменения);
- заблокировать строку (предотвращая ее блокирование другим оператором изменения или удаления).

Как уже упоминалось, это можно сделать независимо от принятого подхода — как при пессимистическом, так и при оптимистическом блокировании можно использовать оператор SELECT **FOR UPDATE NOWAIT** для проверки того, что строка не изменилась. При пессимистическом блокировании этот оператор выполняется в тот момент, когда пользователь выражает намерение изменять данные. При оптимистическом блокировании этот оператор выполняется непосредственно перед изменением данных в базе. Это не только решает проблемы блокирования в приложении, но и обеспечивает целостность данных.

Взаимные блокировки

Взаимные блокировки возникают, когда два сеанса удерживают ресурсы, необходимые другому сеансу. Взаимную блокировку легко продемонстрировать на примере базы данных с двумя таблицами, А и В, в каждой из которых по одной строке. Для этого необходимо начать два сеанса (скажем, два сеанса SQL*Plus) и в сеансе А изменить таблицу А. В сеансе Б надо изменить таблицу В. Теперь, если попытаться изменить таблицу А в сеансе Б, он окажется заблокированным, поскольку соответствующая строка уже заблокирована сеансом А. Это еще не взаимная блокировка — сеанс просто заблокирован. Взаимная блокировка еще не возникла, поскольку есть шанс, что сеанс А зафиксирует или откатит транзакцию и после этого сеанс Б продолжит работу.

Если мы вернемся в сеанс А и попытаемся изменить таблицу В, то вызовем взаимную блокировку. Один из сеансов будет выбран сервером в качестве "жертвы", и в нем произойдет откат оператора. Например, может быть отменена попытка сеанса Б изменить таблицу А с выдачей сообщения об ошибке следующего вида:

```
update a set x = x+1
         *
ERROR at line 1:
ORA-00060: deadlock detected while waiting for resource
```

Попытка сеанса А изменить таблицу В по-прежнему блокируется — сервер Oracle не откатывает всю транзакцию. Откатывается только один из операторов, ставших причиной возникновения взаимной блокировки. Сеанс Б по-прежнему удерживает блокировку строки в таблице В, а сеанс А терпеливо ждет, пока эта строка станет доступной. Получив сообщение о взаимной блокировке, сеанс Б должен решить, фиксировать ли уже выполненные изменения в таблице В, откатить ли их или продолжить работу и зафиксировать транзакцию позднее. Как только этот сеанс зафиксирует или откатит транзакцию, другие заблокированные сеансы смогут продолжить работу.

Для сервера Oracle взаимная блокировка — настолько редкий, необычный случай, что при каждом ее возникновении создается трассировочный файл. Содержимое трассировочного файла примерно таково:

```
*** 2001-02-23 14:03:35.041
*** SESSION ID:(8.82) 2001-02-23 14:03:35.001
DEADLOCK DETECTED
Current SQL statement for this session:
update a set x = x+1
The following deadlock is not an ORACLE error. It is a
deadlock due to user error in the design of an application
or from issuing incorrect ad-hoc SQL. The following...
```

Очевидно, сервер Oracle воспринимает взаимные блокировки как ошибки приложения, и в большинстве случаев это справедливо. В отличие от многих других реляционных СУБД, взаимные блокировки настолько редко происходят в Oracle, что вполне можно игнорировать их существование. Обычно взаимную блокировку необходимо создавать искусственно.

Как свидетельствует опыт, основной причиной возникновения взаимных блокировок в базах данных Oracle являются неиндексированные внешние ключи. При изменении главной таблицы сервер Oracle полностью блокирует подчиненную таблицу в двух случаях:

- при изменении первичного ключа в главной таблице (что бывает крайне редко, если следовать принятому в реляционных базах данных правилу неизменности первичных ключей) подчиненная таблица блокируется при отсутствии индекса по внешнему ключу;
- при удалении строки в главной таблице подчиненная таблица также полностью блокируется (при отсутствии индекса по внешнему ключу).

Чтобы продемонстрировать первый случай, создадим пару таблиц следующим образом:

```
tkyte@TKYTE816> create table p  (x int primary key) ;
Table created.

tkyte@TKYTE816> create table c  (y references p) ;
Table created.

tkyte@TKYTE816> insert into p values  (1) ;
tkyte@TKYTE816> insert into p values  (2) ;

tkyte@TKYTE816> commit;
```

А затем выполним:

```
tkyte@TKYTE816> update p set x = 3 where x = 1;
1 row updated.
```

В результате сеанс заблокировал таблицу C, и никакой другой сеанс не может удалять, вставлять или изменять в ней строки. Повторю еще раз: изменение первичного ключа не приветствуется при работе с реляционными базами данных, так что обычно подобная проблема не возникает. Зато проблема изменения первичного ключа становится актуальной в случае использования средств автоматической генерации SQL-операторов, которые обновляют значения всех столбцов, независимо от того, изменил ли значение пользователь. Например, при создании в Oracle Forms стандартной формы для просмотра и редактирования таблицы по умолчанию генерируется оператор **UPDATE,** изменяющий все просматриваемые столбцы таблицы. Если создать стандартную форму для таблицы **DEPT** и включить в нее все три поля, Oracle Forms будет выполнять следующую команду при изменении **любого** из столбцов таблицы DEPT:

```
update dept set deptno=:1,dname=:2,loc=:3 where rowid=:4
```

В этом случае, если таблица **EMP** имеет внешний ключ, ссылающийся на DEPT, и по столбцу DEPTNO в таблице **EMP** нет индекса, вся таблица **EMP** будет заблокирована после изменения таблицы **DEPT.** За этим надо внимательно следить при использовании любого средства, автоматически генерирующего SQL-операторы. Хотя значение первичного ключа не изменилось, подчиненная таблица **EMP** после выполнения приведенного выше SQL-оператора будет заблокирована. В случае Oracle Forms необ-

ходимо установить значение **Yes** для свойства таблицы **update changed columns only** (обновлять только измененные столбцы). В результате Oracle Forms будет генерировать оператор UPDATE, включающий только изменившиеся столбцы (но не первичный ключ).

Проблемы, связанные с удалением строки в главной таблице, возникают намного чаще. Если удаляется строка в таблице P, то вся подчиненная таблица C оказывается заблокированной, что не позволяет выполнять другие изменения таблицы C в течение всей транзакции (предполагается, конечно, что ни один другой сеанс не изменял таблицу C в момент удаления, иначе оператору удаления пришлось бы ожидать). Именно так возникают проблемы блокирования, в том числе взаимного. Блокирование таблицы C ограничивает возможность одновременной работы с базой данных, — любые изменения в ней становятся невозможными. Кроме того, увеличивается вероятность взаимного блокирования, поскольку сеанс в течение транзакции "владеет" слишком большим объемом данных. Вероятность того, что другой сеанс окажется заблокированным при попытке изменения таблицы C, теперь намного больше. Вследствие этого блокируется множество сеансов, удерживающих определенные ресурсы. Если какой-либо из заблокированных сеансов удерживает ресурс, необходимый исходному, удалившему строку сеансу, возникает взаимная блокировка. Причина взаимной блокировки в данном случае — блокирование исходным сеансом намного большего количества строк, чем реально необходимо. Если кто-то жалуется на взаимные блокировки в базе данных, я предлагаю выполнить сценарий, который находит неиндексированные внешние ключи, и в девяноста девяти процентах случаев мы обнаруживаем таблицу, вызвавшую проблемы. После индексирования соответствующего внешнего ключа взаимные блокировки и множество других конфликтов при доступе исчезают навсегда. Вот пример того, как автоматически находить неиндексированные внешние ключи:

```
tkyte@TKYTE816> column columns format a30 word_wrapped
tkyte@TKYTE816> column tablename format a15 word_wrapped
tkyte@TKYTE816> column constraint_name format a15 word_wrapped

tkyte@TKYTE816> select table_name, constraint_name,
  2        ename1 || nvl2(cname2,','||cname2,null) ||
  3    nvl2(cname3,','||cname,null)    ||    nvl2(cname4,','||cname4,null) ||
  4     nvl2(cname5,','||cname5,null)     ||  nvl2(cname6,','||cname6,null) ||
  5    nvl2(cname7,','||cname7,null)    ||    nvl2(cname8,','||cname8,null)
  6               columns
  7     from (select b.table_name,
  8                  b.constraint_name,
  9                max(decode(position, 1, colunm_name, null)) ename1,
 10                 max(decode(position, 2, column_name, null)) cname2,
 11                 max(decode(position, 3, column_name, null)) cname3,
 12                 max(decode(position, 4, column_name, null)) cname4,
 13                 max(decode(position, 5, colunm_name, null)) cname5,
 14                 max(decode(position, 6, column_name, null)) cname6,
 15                 max(decode(position, 7, column_name, null)) cname7,
 16                 max(decode(position, 8, column_name, null)) cname8,
 17                 count(*) col_cnt
 18            from (select substr(table_name,1,30) table_name,
 19                         substr(constraint_name,1,30) constraint_name,
```

```
20              substr(column_naine,1,30)              column_name,
21                        position
22                    from user_cons_columns) a,
23                  user_constraints b
24             where a.constraint_name = b.constraint_name
25               and b.constraint_type = 'R'
26             group by b.table_name, b.constraint_name
27         ) cons
28   where col_cnt > ALL
29            (select count(*)
30                 from user_ind_columns i
31                where i.table_name = cons.table_name
32                  and i.column_name in (cname1, cname2, cname3, cname4,
33          cname5,           cname6,           cname7,           cname8)
34                  and i.column_position <= cons.col_cnt
35                group by i.index_name
36           )
37  /

TABLE_NAME          CONSTRAINT_NAME      COLUMNS

C                   SYS_C004710          Y
```

Этот сценарий работает с ограничениями внешнего ключа, включающими до 8 столбцов (если у вас используется больше столбцов, имеет смысл пересмотреть структуру). Вначале создается *подставляемое представление* (inline view), названное в данном запросе **CONS.** Это представление транспонирует имена столбцов, входящих в ограничение, представляя их в виде столбцов. В результате для каждого ограничения получается одна строка, включающая до 8 имен столбцов в ограничении. Кроме того, в строке имеется столбец **COL_CNT,** содержащий количество столбцов в самом ограничении внешнего ключа. Для каждой строки, выбранной из представления **CONS,** мы выполняем коррелированный подзапрос, проверяющий все индексы по обрабатываемой таблице. Он считает столбцы в этом индексе, соответствующие столбцам в ограничении внешнего ключа, а затем группирует результаты по имени индекса. Таким образом, он генерирует набор чисел, каждое из которых представляет собой количество совпавших столбцов в одном из индексов таблицы. Если исходное значение в столбце **COL_CNT** больше, чем **все** эти числа, значит, нет ни одного индекса таблицы, обеспечивающего выполнение ограничения внешнего ключа. Если значение в столбце **COL_CNT** меньше некоторых чисел, значит, имеется хотя бы один индекс, обеспечивающий выполнение ограничения. Обратите внимание на использование функции **NVL2** (впервые появилась в Oracle 8.15), которая формирует из имен столбцов список (через запятую). Эта функция принимает три аргумента: A, B и C. Если аргумент A — не пустой, она возвращает аргумент B, в противном случае — аргумент C. В запросе предполагается, что владелец ограничения является также владельцем таблицы и индекса. Если таблица проиндексирована другим пользователем или принадлежит другому пользователю, сценарий будет работать некорректно (но подобные случаи встречаются редко).

Итак, представленный выше сценарий показывает, что таблица C имеет внешний ключ по столбцу Y, но по этому столбцу нет индекса. Проиндексировав столбец Y, мы

можем полностью устранить проблему блокирования. Помимо блокирования таблицы неиндексированный внешний ключ может вызывать проблемы в следующих случаях:

- Если установлено ограничение **ON DELETE CASCADE** и подчиненная таблица не проиндексирована. Например, таблица **EMP** является подчиненной по отношению к **DEPT.** Оператор **DELETE DEPTNO = 10** должен вызвать каскадное удаление (CASCADE) соответствующих строк в таблице **EMP.** Если столбец **DEPTNO** в таблице **EMP** не проиндексирован, придется выполнять *полный просмотр* (full table scan) таблицы **EMP.** Полный просмотр обычно нежелателен, поскольку при удалении большого количества строк из главной таблицы приводит к большим расходам времени и ресурсов.
- При выполнении запроса из главной таблицы в подчиненную. Вернемся опять к примеру с таблицами **EMP** и **DEPT.** Очень часто таблица **EMP** запрашивается с учетом **DEPTNO.** Если постоянно для генерации отчета выполняется, например, следующий запрос, то отсутствие индекса замедлит работу:

```
select * from dept, emp
where emp.deptno = dept.daptno and dept.deptno = :X;
```

Итак, когда индекс по внешнему ключу не нужен? В общем случае, когда выполнены следующие условия:

- не удаляются строки из главной таблицы;
- не изменяется значение уникального/первичного ключа главной таблицы (следите за непреднамеренным обновлением первичного ключа используемыми инструментальными средствами!);
- не выполняется соединение главной и подчиненной таблиц (как в случае с DEPT и **EMP).**

Если соблюдены все три условия, индекс можно не создавать: он не нужен. Если выполняется одно из перечисленных действий, помните о последствиях. Это один из немногих случаев, когда сервер Oracle "избыточно" блокирует данные.

Эскалация блокирования

Когда происходит *эскалация блокирования,* система увеличивает размер блокируемых объектов. Примером может служить блокирование системой всей таблицы вместо 100 отдельных ее строк. В результате одной блокировкой удерживается намного больше данных, чем перед эскалацией. Эскалация блокирования часто используется в СУБД, когда требуется избежать лишнего расходования ресурсов.

В СУБД Oracle **никогда** не применяется эскалация блокирования, однако выполняется *преобразование блокировок* (lock conversion) или *распространение блокировок* (lock promotion). Эти термины часто путают с эскалацией блокирования.

Термины "преобразование блокировок" и "распространение блокировок" — синонимы. В контексте Oracle обычно говорят о преобразовании.

Берется блокировка самого низкого из возможных уровней (наименее ограничивающая блокировка) и преобразуется к более высокому (ограничивающему) уровню. Например, при выборе строки из таблицы с конструкцией FOR UPDATE будет создано две блокировки. Одна из них устанавливается на выбранную строку (или строки); это — исключительная блокировка: ни один сеанс уже не сможет заблокировать соответствующие строки в исключительном режиме. Другая блокировка, ROW SHARE TABLE (совместное блокирование строк таблицы), устанавливается на соответствующую таблицу. Это предотвратит исключительную блокировку таблицы другими сеансами и, следовательно, возможность изменения, например, структуры таблицы. Все остальные операторы смогут работать с таблицей. Другой сеанс может даже сделать таблицу доступной только для чтения с помощью оператора LOCK TABLE X IN SHARE MODE, предотвратив тем самым ее изменение. Однако этот другой сеанс не должен иметь права предотвращать изменения, которые уже происходят. Поэтому, как только будет выполнена команда фактического изменения строки, сервер Oracle преобразует блокировку ROW SHARE TABLE в более ограничивающую блокировку ROW EXCLUSIVE TABLE, и изменение будет выполнено. Такое преобразование блокировок происходит само собой независимо от приложений.

Эскалация блокировок — не преимущество базы данных. Это — нежелательное свойство. Тот факт, что СУБД поддерживает эскалацию блокировок, означает, что ее механизм блокирования расходует слишком много ресурсов, что особенно ощутимо при управлении сотнями блокировок. В СУБД Oracle расходы ресурсов в случае одной или миллиона блокировок одинаковы, — ресурсы просто не расходуются.

Типы блокировок

Ниже перечислены пять основных классов блокировок в Oracle. Первые три — общие (используются во всех базах данных Oracle), а две остальные — только в OPS (Oracle Parallel Server — параллельный сервер). Специфические блокировки OPS мы рассмотрим лишь вкратце, зато общие блокировки — очень подробно.

- *Блокировки ЯМД* (DML locks). ЯМД означает язык манипулирования данными (Data Manipulation Language), т.е. операторы SELECT, INSERT, UPDATE и DELETE. К блокировкам ЯМД относятся, например, блокировки строки данных или блокировка на уровне таблицы, затрагивающая все строки таблицы.
- *Блокировки ЯОД* (DDL locks). ЯОД означает язык определения данных (Data Definition Language), т.е. операторы CREATE, ALTER и так далее. Блокировки ЯОД защищают определения структур объектов.
- *Внутренние блокировки* (internal locks) и *защелки* (latches). Это блокировки, используемые сервером Oracle для защиты своих внутренних структур данных. Например, разбирая запрос и генерируя оптимизированный план его выполнения, сервер Oracle блокирует с помощью защелки библиотечный кэш, чтобы поместить в него этот план для использования другими сеансами. Защелка — это простое низкоуровневое средство обеспечения последовательности обращений, используемое сервером Oracle, и по функциям аналогичное блокировке.

- *Распределенные блокировки* (distributed locks). Эти блокировки используются сервером OPS для согласования ресурсов машин, входящих в кластер. Распределенные блокировки устанавливаются экземплярами баз данных, а не отдельными транзакциями.
- *Блокировки параллельного управления кэшем* (PCM — Parallel Cache Management Locks). Такие блокировки защищают блоки данных в кэше при использовании их несколькими экземплярами.

Теперь мы подробно рассмотрим отдельные типы блокировок в каждом из этих общих классов, в том числе последствия их использования. Существуют и другие типы блокировок, однако они выходят за рамки тем, рассматриваемых в этой книге. Мы рассмотрим блокировки наиболее часто используемые и долго удерживаемые. Блокировки других типов обычно устанавливаются лишь на очень непродолжительное время.

Блокировки ЯМД

Блокировки ЯМД позволяют гарантировать, что в каждый момент времени только одному сеансу позволено изменять строку и что не может быть удалена таблица, с которой работает сеанс. Сервер Oracle автоматически, более или менее прозрачно для пользователей, устанавливает эти блокировки по ходу работы.

TX - блокировки транзакций

Блокировка TX устанавливается, когда транзакция инициирует первое изменение, и удерживается до тех пор, пока транзакция не выполнит оператор **COMMIT** или **ROLLBACK.** Она используется как механизм организации очереди для сеансов, ожидающих завершения транзакции. Каждая измененная или выбранная с помощью SELECT **FOR UPDATE** строка будет "указывать" на соответствующую блокировку TX. Казалось бы, это должно повлечь большие расходы ресурсов, но на самом деле этого не происходит. Чтобы понять, почему, необходимо разобраться, где "живут" блокировки и как сервер ими управляет. В Oracle блокировки хранятся как атрибут данных (см. описание формата блока Oracle в главе 2). У сервера Oracle нет традиционного диспетчера блокировок, поддерживающего длинный список со всеми строками, заблокированными в системе. Другие СУБД делают именно так, поскольку для них блокировки — дорогостоящий ресурс, за использованием которого надо следить. Чем больше блокировок, тем сложнее ими управлять, отсюда и заботы о том, "не слишком ли много" блокировок используется в системе.

Если бы сервер Oracle имел традиционный диспетчер блокировок, при блокировании строки нужно было бы выполнить примерно такую последовательность действий.

1. Найти адрес строки, которую необходимо заблокировать.
2. Подключиться к диспетчеру блокировок (необходимо выполнять по очереди, поскольку используются общие структуры в памяти.)
3. Заблокировать список.
4. Просмотреть список, чтобы проверить, не заблокирована ли эта строка другим сеансом.

5. Создать в списке новую запись, фиксирующую факт блокирования строки.
6. Разблокировать список.

Теперь, когда строка заблокирована, ее можно изменять. При последующей фиксации изменений необходимо:

7. Снова подключиться к диспетчеру.
8. Заблокировать список блокировок.
9. Найти в списке и снять все установленные блокировки.
10. Разблокировать список.

Как видите, чем больше установлено блокировок, тем больше времени потребуется для изменения данных и фиксации этих изменений. Поэтому сервер Oracle поступает примерно так:

1. Находит адрес строки, которую необходимо заблокировать.
2. Переходит на эту строку.
3. Блокирует ее (ожидая снятия блокировки, если она уже заблокирована и при этом не используется опция NOWAIT).

Вот и все. Поскольку блокировка хранится как атрибут данных, серверу Oracle не нужен традиционный диспетчер блокировок. Транзакция просто переходит к соответствующим данным и блокирует их (если они еще не заблокированы). Иногда при обращении данные кажутся заблокированными, хотя фактически они уже не заблокированы. При блокировании строки данных в Oracle с блоком данных связывается идентификатор транзакции, причем остается там после снятия блокировки. Этот идентификатор уникален для нашей транзакции и задает номер сегмента отката, слот и номер изменения (sequence number). Оставляя его в блоке, содержащем измененную строку, мы как бы говорим другим сеансам, что "эти данные принадлежат нам" (не все данные в блоке, только одна строка, которую мы меняем). Когда другой сеанс обращается к блоку, он "видит" идентификатор блокировки и, "зная", что он представляет транзакцию, определяет, активна ли еще транзакция, установившая блокировку. Если транзакция уже закончена, сеанс может получить данные "в собственность". Если же транзакция активна, сеанс "попросит" систему уведомить его о завершении транзакции. Таким образом, имеется механизм организации очереди: сеанс, нуждающийся в блокировке, будет помещен в очередь в ожидании завершения транзакции, после чего получит возможность работать с данными.

Вот небольшой пример, демонстрирующий, как это происходит. Чтобы увидеть, как работает сервер, используют три таблицы* динамической производительности V$:

- V$TRANSACTION, содержащую запись о каждой активной транзакции;
- V$SESSION, которая показывает действующие сеансы;
- V$LOCK, содержащую записи обо всех блокировках, с указанием сеансов, ожидающих их снятия.

На самом деле это представления, но мы вслед за автором будем называть их таблицами. *Прим. иаучн. ред.

Во-первых, начнем транзакцию:

```
tkyte@TKYTE816> update dept set deptno = deptno+10;
4 rows updated.
```

Теперь рассмотрим состояние системы в этот момент:

```
tkyte@TKYTE816> select username,
  2           v$lock.sid,
  3      trunc(id1/power(2,16))      rbs,
  4      bitand(id1,power(2,16)-1)+0      slot,
  5          id2 seq,
  6           lmode,
  7           request
  8   from v$lock, v$session
  9   where v$lock.type = 'TX'
 10     and v$lock.sid = v$session.sid
 11     and v$session. username = USER
 12  /

USERNAME      SID      RBS     SLOT      SEQ     LMODE    REQUEST

TKYTE           8        2       46      160         6          0

tkyte@TKYTE816> select XIDUSM, XIDSLOT, XIDSQN
  2     from v$transaction
  3   /

    XIDUSN  XIDSLOT    XIDSQN

         2       46      160
```

Обратите внимание на следующее:

- Значение столбца **LMODE в** таблице **V$LOCK** равно 6, а значение столбца REQUEST — 0. Обратившись к описанию таблицы **V$LOCK** в руководстве *Oracle Server Reference Manual,* можно узнать, что **LMODE=6** означает исключительную блокировку. Значение 0 в столбце **REQUEST** означает, что мы не запрашиваем блокировку, а удерживаем ее.

- В таблице только одна строка. Таблица **V$LOCK** содержит не сами блокировки, а очередь на них. Можно ожидать найти в таблице **V$LOCK** четыре строки, поскольку мы заблокировали именно такое количество строк. Надо, однако, помнить, что сервер Oracle не хранит список заблокированных строк. Чтобы узнать, не заблокирована ли строка, надо ее прочитать.

- Возьмем значения столбцов **ID1** и **ID2** и выполним с ними определенную манипуляцию. Серверу Oracle необходимо хранить три 16-битовых числа, но для этого имеется только два столбца. Поэтому первый столбец, **ID1,** хранит два из этих чисел. Разделив значение столбца на 2^16 с использованием конструкции **trunc(id1/power(2,16)) rbs** и замаскировав старшие биты с помощью **bitand(id1,power(2,16)-1)+0 slot,** получаем два числа, хранящиеся в виде одного значения.

- Значения RBS, SLOT и SEQ сопоставляются с информацией в таблице V$TRANSACTION. Это — идентификатор нашей транзакции.

Теперь откроем другой сеанс от имени того же пользователя, поменяем ряд строк в таблице EMP, а затем попытаемся изменить таблицу DEPT:

```
tkyte@TKYTE816> update emp set ename = upper(ename);
14 rows updated.

tkyte@TKYTE816> update dept set deptno = deptno-10;
```

Сеанс окажется заблокированным. Выполнив запросы к таблицам V$ снова, можно увидеть следующее:

```
tkyte@TKYTE816> select username,
  2           v$lock.sid,
  3                trunc(idl/power(2,16))  rba,
  4         bitand(id1,power(2,16)-1)+0         slot,
  5                id2  seq,
  6           lmode,
  7                request
  8   from v$lock,  v$session
  9   where v$lock.type =  'TX'
 10     and v$lock.sid = v$session.aid
 11     and vSsession.username = USER
 12  /

USERNAME     SID       RBS      SLOT      SEQ     LMODE    REQUEST

TKYTE          8         2        46      160         6          0
TKYTE          9         2        46      160         0          6
TKYTE          9         3        82      163         6          0

tkyte@TKYTE816> select XIDUSN,  XIDSLOT,  XIDSQN
  2     from v$transaction
  3   /

   XIDUSN    XIDSLOT     XIDSQN

        3         82        163
        2         46        160
```

Итак, понятно, что начата новая транзакция с идентификатором (3,82,163). Наш сеанс, **SID=9,** теперь имеет две строки в таблице **V$LOCK.** Одна строка представляет удерживаемые им блокировки (та, где **LMODE=6).** Там же есть строка, содержащая значение 6 в столбце REQUEST. Это запрос исключительной блокировки. Интересно отметить, что значения **RBS/SLOT/SEQ** в этой строке запроса представляют идентификатор транзакции сеанса, *удерживающего* блокировку. Транзакция сеанса со значением SID=8 блокирует транзакцию сеанса со значением **SID=9.** Это можно явно увидеть, выполнив соединение таблицы **V$LOCK** с самой собой:

```
tkyte@TKYTE816> select
              (select username from v$session where sid=a.sid) blocker,
   2           a.sid,
   3         ' is blocking ',
   4           (select username from v$session where sid=b.sid) blockee,
   5               b.sid
   6     from v$lock a, v$lock b
   7    where a.block = 1
   8      and b.request > 0
   9      and a.id1 = b.id1
  10      and a.id2 = b.id2
  11  /

BLOCKER       SID  'ISBLOCKING'   BLOCKEE    SID

TKYTE           8  is blocking   TKYTE        9
```

Теперь, если зафиксировать исходную транзакцию в сеансе SID=8 и выполнить запрос повторно, окажется, что строка запроса блокировки исчезла:

```
tkyte@TKYTE816> select username,
   2           v$lock.sid,
   3         trunc(id1/power(2,16))         rbs,
   4         bitand(id1,power(2,16)-1)+0         slot,
   5              id2 seq,
   6           lmode,
   7               request, block
   8   from v$lock, v$session
   9   where v$lock.type = 'TX'
  10     and v$lock.aid = v$session.sid
  11     and v$session. username = USER
  12  /

USERNAME       SID      RBS     SLOT      SEQ     LMODE    REQUEST

TKYTE            9        3       82      163         6          0
tkyte@TKYTE816> select XIDUSN, XIDSLOT, XIDSQN
   2     from v$transaction
   3  /

   XIDUSN  XIDSLOT    XIDSQN

        3      82       163
```

Строка запроса блокировки, использовавшаяся для организации очереди, исчезает одновременно со снятием блокировки другим сеансом. Сервер может "разбудить" заблокированные сеансы в момент завершения транзакции.

Можно бесконечно "улучшать" представление данных с помощью различных графических средств, но важнее знать, в какие таблицы смотреть.

Напоследок необходимо разобраться еще в одном: как хранится информация о блокировках и транзакциях в самих данных. Она является частью служебной информации блока. В главе 2 было описано, что базовый формат предусматривает в начале блока

"системное" пространство для хранения таблицы транзакций для этого блока. Такая таблица транзакций включает записи для каждой "реальной" транзакции, заблокировавшей те или иные данные в этом блоке. Размер этой структуры управляется двумя атрибутами хранения, задаваемыми в операторе CREATE при создании объекта:

- INITRANS — первоначальный, заранее заданный размер этой структуры. Стандартное значение: 2 — для индексов и 1 — для таблиц.
- MAXTRANS — максимальный размер, до которого может разрастаться эта структура. Стандартное значение: 255.

Итак, каждый блок начинает стандартно свое существование с одним или двумя слотами для транзакций. Количество активных транзакций, которые могут одновременно работать с блоком, ограничивается значением MAXTRANS и доступностью пространства в блоке. Невозможно достичь 255 одновременных транзакций в блоке, если там нет места для роста данной структуры.

Можно искусственно продемонстрировать, как это происходит, создав таблицу с ограниченным значением MAXTRANS. Например:

```
tkyte@TKYTE816> create table t (x int) maxtrans 1;
Table created.

tkyte@TKYTE816> insert into t values (1);
1 row created.

tkyte@TKYTE816> insert into t values (2);
1 row created.

tkyte@TKYTE816> commit;
Commit complete.
```

Теперь в одном сеансе выполняем:

```
tkyte@TKYTE816> update t set x - 3 where x = 1;
1 row updated.
```

а в другом:

```
tkyte@TKYTE816> update t set x = 4 where x = 2;
```

Поскольку обе строки находятся в одном и том же блоке и параметру MAXTRANS (максимальная степень одновременности доступа для данного блока) установлено значение 1, второй сеанс будет заблокирован. Это демонстрирует, что происходит, когда транзакции в количестве, превышающем значение параметра MAXTRANS, пытаются одновременно обратиться к одному и тому же блоку. Точно так же происходит блокирование, если параметр INITRANS имеет небольшое значение и в блоке нет места для динамического добавления транзакции. В большинстве случаев стандартные значения 1 и 2 для параметра INITRANS приемлемы, поскольку таблица транзакций будет динамически расти (в пределах имеющегося свободного пространства), но в некоторых средах для повышения параллелизма и уменьшения количества ожидающих сеансов это значение придется увеличить. В качестве примера можно привести таблицу или, ско-

рее, индекс (поскольку в блоках индекса может храниться намного больше строк, чем в обычном блоке таблицы), который часто изменяется. Может понадобиться увеличить значение INITRANS, чтобы заранее зарезервировать достаточно места в блоке для предполагаемого количества одновременных транзакций. Это особенно необходимо, если предполагается заполнение блоков практически целиком и может не остаться места для динамического расширения таблицы транзакций в блоке.

TM — блокировки очередности ЯМД

Такие блокировки позволяют быть уверенным, что структура таблицы не изменится при изменении ее содержимого. Рассмотрим такой пример. При изменении таблицы на нее устанавливается блокировка TM; это предотвращает применение к ней операторов DROP или ALTER другим сеансом. Если сеанс пытается применить оператор ЯОД к таблице, на которую другой сеанс установил блокировку, он получит следующее сообщение об ошибке:

```
drop table dept
           *
ERROR at line 1:
ORA-00054: resource busy and acquire with NOWAIT specified
```

Это сообщение поначалу сбивает с толку, поскольку нет никакого способа задать конструкции NOWAIT или WAIT в операторе DROP TABLE. Это просто сообщение общего вида, которое сеанс получает, пытаясь выполнить операцию, которая будет заблокирована, хотя не допускает блокирования. Как мы уже видели, такое же сообщение отображается при попытке применить оператор SELECT FOR UPDATE NOWAIT к заблокированной строке.

Ниже показано, как эти блокировки будут отражены в таблице V$LOCK:

```
tkyte@TKYTE816> create table t1 (x int) ;
Table created.

tkyte@TKYTE816> create table t2 (x int) ;
Table created.

tkyte@TKYTE816> insert into t1 values (1);
1 row created.

tkyte@TKYTE816> insert into t2 values (1);
1 row created.

tkyte@TKYTE816> select username,
  2           v$lock.sid,
  3        id1,        id2,
  4           lmode,
  5               request, block, v$lock.type
  6  from v$lock, v$session
  7  where v$lock.sid = v$session.sid
  8    and v$session.username = USER
  9  /
```

```
USERNAME        SID       ID1       ID2      LMODE     REQUEST     BLOCK  TY

TKYTE             8     24055         0          3           0         0  TM
TKYTE             8     24054         0          3           0         0  TM

TKYTE             8    327697       165          6           0         0  TX

tkyte@TKYTE816> select object_name, object_id from user_objects;

OBJECT_NAME                 OBJECT_ID

T1                              24054
T2                              24055
```

Хотя для каждой транзакции может быть только одна блокировка TX, можно устанавливать столько блокировок TM, сколько объектов изменяется. В этом случае значение столбца **ID1** для блокировки TM представляет собой идентификатор объекта, заблокированного оператором ЯМД, так что найти блокируемый объект легко.

Интересная особенность блокировки TM: общее количество блокировок TM, поддерживаемых в системе, конфигурируется администратором (подробнее см. описание параметра **DML_LOCKS** файла **init.ora** в руководстве *Oracle8i Server Reference)*. Его можно даже установить равным нулю. Это не означает, что база данных становится доступной только для чтения (так как блокировки не поддерживаются), — в ней не разрешены операторы ЯОД. Это используется, например, в сервере OPS для упрощения координации экземпляров. Можно также лишить возможности получать блокировки TM для отдельных объектов с помощью оператора **ALTER TABLE имя_таблицы DISABLE TABLE LOCK.**

Блокировки ЯОД

Блокировки ЯОД автоматически устанавливаются на объекты в ходе выполнения операторов ЯОД для защиты их от изменения другими сеансами. Например, при выполнении оператора ЯОД **ALTER TABLE T** на таблицу T будет установлена исключительная блокировка ЯОД, что предотвращает установку блокировок ЯОД и TM на эту таблицу другими сеансами. Блокировки ЯОД удерживаются на период выполнения оператора ЯОД и снимаются сразу по его завершении. Это делается путем помещения операторов ЯОД в неявные пары операторов фиксации (или фиксации и отката). Вот почему операторы ЯОД в Oracle всегда фиксируются. Операторы **CREATE, ALTER** и т.д. фактически выполняются, как показано в следующем псевдокоде:

```
Begin
    Commit;
    Оператор ЯОД
    Commit;
Exception
    When others then rollback;
End;
```

Поэтому операторы ЯОД всегда фиксируют транзакцию, даже если завершаются неудачно. Выполнение оператора ЯОД начинается с фиксации. Помните об этом. Сна-

чала выполняется фиксация, чтобы в случае отката не пришлось откатывать предыдущую часть транзакции. При выполнении оператора ЯОД фиксируются все выполненные ранее изменения, даже если сам оператор ЯОД выполнен неудачно. Если должен быть выполнен оператор ЯОД, но не требуется, чтобы он зафиксировал существующую транзакцию, можно использовать автономную транзакцию (подробнее см. в главе 15).

Имеется три типа блокировок ЯОД:

- **Исключительные блокировки ЯОД.** Они предотвращают установку блокировок ЯОД или ТМ (ЯМД) другими сеансами. Это означает, что можно запрашивать таблицу в ходе выполнения оператора ЯОД, но нельзя ее изменять.
- **Разделяемые блокировки ЯОД.** Они защищают структуру соответствующего объекта от изменения другими сеансами, но разрешают изменять данные.
- **Нарушаемые блокировки разбора** (breakable parse locks). Они позволяют объекту, например плану запроса, хранящемуся в кэше разделяемого пула, зарегистрировать свою зависимость от другого объекта. При выполнении оператора ЯОД, затрагивающего заблокированный таким образом объект, сервер Oracle получает список объектов, зарегистрировавших свою зависимость, и помечает их как недействительные. Вот почему эти блокировки — "нарушаемые": они не предотвращают выполнение операторов ЯОД.

Большинство операторов ЯОД устанавливает *исключительную* блокировку ЯОД. При выполнении оператора, подобного

```
Alter table t add new_column date;
```

таблица T будет недоступна для изменения, пока оператор выполняется. К таблице в этот период можно обращаться с помощью оператора SELECT, но другие действия, в том числе операторы ЯОД, блокируются. В Oracle 8i некоторые операторы ЯОД теперь могут выполняться без установки блокировок ЯОД. Например, можно выполнить:

```
create index t_idx on t(x) ONLINE;
```

Ключевое слово **ONLINE** изменяет метод построения индекса. Вместо установки исключительной блокировки ЯОД, предотвращающей изменения данных, Oracle попытается установить на таблицу низкоуровневую (режим 2) блокировку ТМ. Это предотвращает изменения структуры с помощью операторов ЯОД, но позволяет нормально выполнять операторы ЯМД. Сервер Oracle достигает этого путем записи в таблице изменений, сделанных в ходе выполнения оператора ЯОД, и учитывает эти изменения в новом индексе, когда завершается его создание. Это существенно увеличивает доступность данных.

Другие типы операторов ЯОД устанавливают *разделяемые* блокировки ЯОД. Они устанавливаются на объекты, от которых зависят скомпилированные хранимые объекты, типы процедур и представлений. Например, если выполняется оператор:

```
Create view MyView
as
select *
  from emp, dept
 where emp. deptno = dept.deptno;
```

разделяемые блокировки ЯОД будут устанавливаться на таблицы EMP и DEPT на все время выполнения оператора **CREATE VIEW.** Мы можем изменять содержимое этих таблиц, но не их структуру.

Последний тип блокировок ЯОД — *нарушаемые блокировки разбора.* Когда сеанс разбирает оператор, блокировка разбора устанавливается на каждый объект, упоминаемый в этом операторе. Эти блокировки устанавливаются, чтобы разобранный и помещенный в кэш оператор был признан недействительным (и выброшен из кэша в разделяемой памяти), если один из упоминаемых в нем объектов удален или изменена его структура.

При поиске этой информации особо ценным будет представление **DBA_DDL_LOCKS.** (Ни одного походящего для этого представления V$ не существует). Представление **DBA_DDL_LOCKS** строится по более "мистическим" таблицам X$ и по умолчанию не создается в базе данных. Для установки его и других представлений, связанных с блокировками, выполните сценарий **CATBLOCK.SQL** из каталога **[ORACLE_HOME]/rdbms/admin.** Этот сценарий можно успешно выполнить от имени пользователя SYS. После выполнения этого сценария можно выполнять запросы к указанному представлению. Например, в базе данных с одним пользователем я получил такой результат:

```
tkyte@TKYTE816> select * from dba_ddl_locks;
session                                                  mode  mode
  id OWNE RNAME                    TYPE                   held  reqe

   8 SYS  DBMS_APPLICATION_INFO    Body                   Null  None
   8 SYS  DBMS_APPLICATION_INFO    Table/Procedure/Type   Null  None
   8 SYS  DBMS_OUTPOT              Table/Procedure/Type   Null  None
   8 SYS  DBMS_OUTPUT              Body                   Null  None
   8 TKYTE TKYTE                   18                     Null  None
   8 SYS  DATABASE                 18                     Null  None

6 rows selected.
```

Вот и все объекты, "блокируемые" моим сеансом. Установлены нарушаемые блокировки разбора на два пакета **DBMS_*.** Это — побочный эффект использования утилиты SQL*Plus; она, например, вызывает пакет **DBMS_APPLICATION_INFO. В** результатах может оказаться несколько строк для одного и того же объекта — это нормально и означает, что в разделяемом пуле сеанс использует несколько объектов, ссылающихся на данный объект. Интересно отметить, что в столбце **OWNER** этого представления указан не владелец блокировки, а владелец блокируемого объекта. Вот почему в нескольких строках указан пользователь SYS: он владеет этими пакетами, но блокирует их мой сеанс.

Чтобы понять, как действует нарушаемая блокировка, создадим и выполним хранимую процедуру P:

```
tkyte@TKYTE816> create or replace procedure p as begin null; end;
  2  /
Procedure created.

tkyte@TKYTE816> exec p

PL/SQL procedure successfully completed.
```

Процедура P теперь упоминается в представлении DBA_DDL_LOCKS. На нее установлена блокировка разбора:

```
tkyte@TKYTE816> select * from dba_ddl_locks;

session                                                    mode mode
    id OWNER    NAME                    TYPE               held reqe

     8 TKYTE    P                       Table/Procedure/Type    Null None
     8 SYS      DBMS_APPLICATION_INFO   Body                    Null None
     8 SYS      DBMS_APPLICATION_INFO   TaЫe/Procedure/Type     Null None
     8 SYS      DBMS_OUTPUT             Table/Procedure/Type    Null None
     8 SYS      DBMS_OUTPUT             Body                    Null None
     8 TKYTE    TKYTE                   18                      Null None
     8 SYS      DATABASE                18                      Null None

1 rows selected.
```

Перекомпилируем процедуру и еще раз запросим представление:

```
tkyte@TKYTE816> alter procedure p compile;

Procedure altered.

tkyte@TKYTE816> select * from dba_ddl_locks;

session                                                    mode mode
    id OWNER    NAME                    TYPE               held reqe

     8 SYS      DBMS_APPLICATION_INFO   Body                    Null None
     8 SYS      DBMS_APPLICATION_INFO   TaЫe/Procedure/Type     Null None
     8 SYS      DBMS_OUTPUT             Table/Procedure/Type    Null None
     8 SYS      DBMS_OUTPUT             Body                    Null None
     8 TKYTE    TKYTE                   18                      Null None
     8 SYS      DATABASE                18                      Null None

6 rows selected.
```

Теперь строки для процедуры P в представлении нет — наша нарушаемая блокировка снята.

Это представление пригодится разработчикам, когда обнаружится, что какой-то фрагмент кода не компилируется в тестовой среде или среде разработки — он зависает и в конечном итоге попытка завершается неудачно. Это показывает, что этот код использует (фактически — выполняет) какой-то другой сеанс, и представление позволяет понять, какой именно. То же самое происходит с операторами GRANT и другими типами операторов ЯОД, применяемыми к объекту. Нельзя, например, предоставить право на выполнение (EXECUTE) процедуры, работающей в настоящий момент. Для обнаружения потенциально блокирующих и ожидающих снятия блокировки сеансов можно использовать описанный выше метод.

Защелки и внутренние блокировки

Защелки и *внутренние блокировки* (enqueues) — простейшие средства обеспечения очередности доступа, используемые для координации многопользовательского доступа к общим структурам данных, объектам и файлам.

Защелки — это блокировки, удерживаемые в течение очень непродолжительного времени, достаточного, например, для изменения структуры данных в памяти. Они используются для защиты определенных структур памяти, например буферного кэша или библиотечного кэша в разделяемом пуле (эти структуры описаны в главе 2). Защелки обычно запрашиваются системой в режиме ожидания. Это означает, что, если защелку нельзя установить, запрашивающий сеанс прекращает работу ("засыпает") на короткое время, а затем пытается повторить операцию. Другие защелки могут запрашиваться в оперативном режиме, т.е. процесс будет делать что-то другое, не ожидая возможности установить защелку. Поскольку возможности установить защелку может ожидать несколько запрашивающих сеансов, одни из них будут ожидать дольше, чем другие. Защелки выделяются случайным образом по принципу "кому повезет". Сеанс, запросивший установку защелки сразу после освобождения ресурса, установит ее. Нет очереди ожидающих освобождения защелки — есть просто "толпа" пытающихся ее получить.

Для работы с защелками Oracle использует неделимые инструкции типа "проверить и установить". Поскольку инструкции для установки и снятия защелок — неделимые, операционная система гарантирует, что только один процесс сможет установить защелку. Поскольку это делается одной инструкцией, то происходит весьма быстро. Защелки удерживаются непродолжительное время, причем имеется механизм очистки на случай, если владелец защелки "скоропостижно скончается", удерживая ее. Эта очистка обычно выполняется процессом PMON.

Внутренние блокировки — более сложное средство обеспечения очередности доступа, используемое, например, при изменении строк в таблице базы данных. В отличие от защелок, они позволяют запрашивающему "встать в очередь" в ожидании освобождения ресурса. Запрашивающий защелку сразу уведомляется, возможно ли это. В случае внутренней блокировки запрашивающий блокируется до тех пор, пока не сможет эту блокировку установить. Таким образом, внутренние блокировки работают медленнее защелок, но обеспечивают гораздо большие функциональные возможности. Внутренние блокировки можно устанавливать на разных уровнях, поэтому можно иметь несколько "разделяемых" блокировок и блокировать с разными уровнями "совместности".

Блокирование вручную. Блокировки, определяемые пользователем

До сих пор мы рассматривали в основном блокировки, устанавливаемые сервером Oracle без нашего вмешательства. При изменении таблицы сервер Oracle устанавливает на нее блокировку TM, чтобы предотвратить ее удаление (как фактически и применение к ней большинства операторов ЯОД) другими сеансами. В изменяемых блоках оставляют блокировки TX — благодаря этому другие сеансы "знают", что с данными рабо-

тают. Сервер использует блокировки ЯОД для защиты объектов от изменений по ходу их изменения сеансом. Он использует защелки и внутренние блокировки для защиты собственной структуры. Теперь давайте посмотрим, как включиться в процесс блокирования. У нас есть следующие возможности:

- блокирование данных вручную с помощью оператора SQL;
- создание собственных блокировок с помощью пакета **DBMS_LOCK.**

Рассмотрим, зачем могут понадобиться эти средства.

Блокирование вручную

Мы уже описывали несколько случаев, когда может потребоваться блокирование вручную. Основным методом явного блокирования данных вручную является использование оператора **SELECT...FOR UPDATE.** Мы использовали его в предыдущих примерах для решения проблемы потерянного изменения, когда один сеанс может переписать изменения, сделанные другим сеансом. Мы видели, что этот оператор используется для установки очередности доступа к подчиненным записям, диктуемой бизнес-правилами (вспомните пример с планировщиком ресурсов, приведенный в главе 1).

Данные можно также заблокировать вручную с помощью оператора **LOCK TABLE.** На практике это используется редко в силу особенностей такой блокировки. Этот оператор блокирует таблицу, а не строки в ней. При изменении строк они "блокируются" как обычно. Так что это — не способ экономии ресурсов (как, возможно, в других реляционных СУБД). Оператор **LOCK TABLE IN EXCLUSIVE MODE** имеет смысл использовать при выполнении большого пакетного изменения, затрагивающего множество строк таблицы, и необходима уверенность, что никто не "заблокирует" это действие. Блокируя таким способом таблицу, можно быть уверенным, что все изменения удастся выполнить без блокирования другими транзакциями. Но приложения с оператором **LOCK TABLE** встречаются крайне редко.

Создание собственных блокировок

Сервер Oracle открывает для разработчиков свои механизмы блокирования для обеспечения очередности доступа с помощью пакета **DBMS_LOCK** (который подробно описывается в приложении А). Может показаться странным, зачем вообще создавать собственные блокировки. Ответ обычно зависит от того, какое используется приложение. Например, этот пакет может применяться для обеспечения последовательного доступа к ресурсу, внешнему по отношению к серверу Oracle. Пусть используется подпрограмма пакета **UTL_FILE,** позволяющая записывать информацию в файл в файловой системе сервера. Предположим, разработана общая подпрограмма передачи сообщений, вызываемая приложениями для записи сообщений. Поскольку файл является внешним, сервер Oracle не может координировать доступ нескольких пользователей, пытающихся его менять одновременно. В таких случаях как раз и пригодится пакет **DBMS_LOCK.** Прежде чем открывать файл, записывать в него и закрывать, можно устанавливать блокировку с именем, соответствующим имени файла, в исключительном режиме, а после закрытия файла вручную снимать эту блокировку. В результате только один пользова-

тель в каждый момент времени сможет записывать сообщение в этот файл. Всем остальным придется ждать. Пакет DBMS_LOCK позволяет вручную снять блокировку, когда она уже больше не нужна, или дождаться автоматического ее снятия при фиксации транзакции, или сохранить ее до завершения сеанса.

Что такое управление одновременным доступом?

Средства управления одновременным доступом — это набор функций, предоставляемых сервером баз данных для обеспечения одновременного доступа к данным и их изменения множеством пользователей. Реализация механизма блокирования в базе данных, вероятно, наиболее существенный фактор при определении степени параллелизма, обеспечиваемого приложением (если проще, масштабируемости). Как уже было сказано, есть множество видов блокировок. Это и блокировки транзакций (TX), максимально масштабируемые как по производительности, так и по количеству (неважно, одна блокировка в системе или миллион); и блокировки TM и ЯОД (применяемые, по возможности, в минимально ограничивающем режиме). Это и блокировки, используемые сервером Oracle в процессе управления доступом к общим структурам данных, — от очень эффективного и быстрого механизма защелок до более громоздкого, но полнофункционального механизма внутренних блокировок.

Но управление одновременным доступом связано не только с блокировками. СУБД может предоставлять и другие средства для обеспечения управляемого, но в значительной степени параллельного доступа к данным. Например, в Oracle имеется средство обеспечения многовариантного доступа (описанное в главе 1). Поскольку сервер Oracle использует многовариантный доступ для обеспечения согласованных по чтению представлений данных, мы получаем весьма приятный побочный эффект: сеанс, читающий данные, никогда не будет заблокирован сеансом, записывающим данные, т.е. запись не блокирует чтения. Это одно из фундаментальных отличий Oracle от остальных СУБД. Запрос на чтение в Oracle никогда не блокируется, он никогда не станет причиной взаимной блокировки с другим сеансом и никогда не даст результат, не существующий в базе данных.

Модель многовариантного доступа Oracle для обеспечения согласованности по чтению всегда применяется на уровне оператора (для каждого запроса), но может применяться и на уровне транзакции. В этом разделе я бы хотел продемонстрировать, как многовариантный доступ используется с различными уровнями изолированности транзакции, определяемыми стандартом SQL92.

Уровни изолированности транзакции

Стандарт ANSI/ISO SQL92 определяет четыре уровня изолированности транзакции, дающих разные результаты для одного и того же сценария транзакции. То есть выполняются одни и те же действия, одинаковым способом, с теми же данными, но, в зависимости от уровня изолированности транзакции, результат может быть различным. Эти

уровни изолированности определяются в терминах трех "явлений", допускаемых или запрещаемых на данном уровне изолированности транзакции:

- *Грязное чтение* (dirty read). Результат настолько же плох, как и название. Допускается чтение незафиксированных, или "грязных", данных. Это случается при открытии обычного файла операционной системы, данные в который записываются другим процессом, и чтении содержимого этого файла. При этом нарушается как целостность данных, так и требования внешнего ключа, а требования уникальности игнорируются.
- *Неповторяемость при чтении* (non-REPEATABLE READ). Это означает, что если строка читается в момент времени T1, а затем перечитывается в момент времени T2, то за этот период она может измениться. Строка может исчезнуть, может быть обновлена и *т.д.*
- *Чтение фантомов* (phantom read). Это означает, что если выполнить запрос в момент времени T1, а затем выполнить его повторно в момент времени T2, в базе данных могут появиться дополнительные строки, влияющие на результаты. От неповторяемости при чтении это явление отличается тем, что прочитанные данные не изменились, но критериям запроса стало удовлетворять **больше** данных, чем прежде.

Уровни изолированности SQL92 определяются как допускающие или разрешающие возникновение каждого из описанных выше явлений:

Уровень изолированности	*Грязное чтение*	*Неповторяемость при чтении*	*Чтение фантомов*
READ UNCOMMITTED	Разрешено	Разрешено	Разрешено
READ COMMITTED		Разрешено	Разрешено
REPEATABLE READ			Разрешено
SERIALIZABLE			

Сервер Oracle явно поддерживает уровни изолированности **READ COMMITTED** и **SERIALIZABLE,** как они определены стандартом. Однако это еще не все. В стандарте SQL92 была попытка установить уровни изолированности транзакции, обеспечивающие различную степень согласованности для запросов, выполняемых на каждом из уровней. **REPEATABLE READ** — уровень изолированности, по утверждению создателей, гарантирующий получение согласованных по чтению результатов запроса. При реализации в соответствии с их определением **READ COMMITTED** не дает согласованных результатов, а уровень изолированности **READ UNCOMMITTED** используется для не блокирующего чтения.

В Oracle уровень изолированности **READ COMMITTED** позволяет достичь согласованности запросов по чтению. (В других СУБД запросы в транзакции с уровнем **READ COMMITTED** могут возвращать результаты, никогда не существовавшие в базе данных.) Более того, Oracle поддерживает также то, для чего предназначался уровень изолиро-

ванности **READ UNCOMMITTED.** Грязное чтение применяется в качестве не блокирующего, т.е. запросы не блокируются при изменении данных и сами не блокируют изменения читаемых данных. Однако серверу Oracle для этого не нужно грязное чтение (он его и не поддерживает). Грязное чтение — это реализация не блокирующего чтения, которое вынуждены использовать другие СУБД.

Помимо четырех уровней изолированности транзакции, определенных в стандарте SQL92, Oracle обеспечивает еще один уровень — транзакции только для чтения. Транзакция только для чтения эквивалентна при чтении уровню изолированности **REPEATABLE READ** или **SERIALIZABLE** в SQL92. Такая транзакция видит только изменения, зафиксированные на момент ее начала, но в этом режиме не разрешена вставка, изменение и удаление данных (другие сеансы могут изменять данные, но транзакция только для чтения — нет). Используя этот режим, можно достичь уровней **REPEATABLE READ** и **SERIALIZABLE READ** без чтения фантомов.

Теперь давайте рассмотрим, как многовариантный доступ и согласованность по чтению соотносятся с перечисленными выше схемами изолированности и как другие СУБД, не поддерживающие многовариантного доступа, достигают тех же результатов. Это будет инструктаж для пользователей других СУБД и тех, кто верит, будто знает, что должны обеспечивать уровни изолированности. Также будет интересно посмотреть, как стандарт, SQL92, который, казалось бы, должен сглаживать различия между СУБД, фактически их допускает. Этот очень детальный стандарт может быть реализован множеством способов.

Уровень изолированности READ UNCOMMITTED

Уровень изолированности **READ UNCOMMITTED** разрешает грязное чтение. Сервер Oracle не использует грязного чтения, и не допускает его. Основное назначение уровня изолированности **READ UNCOMMITTED** — стандартное определение не блокирующего чтения. Как уже было показано, Oracle по умолчанию обеспечивает не блокирующее чтение. Надо создать специальные условия, чтобы оператор **SELECT** блокировал что-либо в базе данных (есть частный случай сомнительной распределенной транзакции, который рассматривается в главе 4). Каждый отдельный оператор, будь-то **SELECT, INSERT, UPDATE** или **DELETE,** выполняется как согласованный по чтению.

Реализуемый сервером Oracle способ достижения согласованности по чтению был продемонстрирован в главе 1 на примере банковских счетов. Здесь мы снова вернемся к этому примеру, чтобы более детально описать, что происходит в СУБД Oracle при многовариантном доступе, а что — в других СУБД. Напомню, что в блоке данных предполагалось наличие одной строки таблицы.

Начнем с той же простой таблицы и запроса:

```
create table accounts (
  account_number number primary key,
  account_balance number
);
select sum(account balance) from accounts;
```

Перед началом запроса имеются следующие данные:

Строка	*Номер счета*	*Баланс счета*
1	123	500,00 $
2	456	240,25 $
...	...	...
342023	987	100,00 $

Теперь оператор **SELECT** начинает выполняться и читать строку 1, строку 2 и так далее. В какой-то момент выполнения запроса некая транзакция переводит 400,00 $ со счета 123 на счет 987. Эта транзакция делает два изменения, но не фиксирует их. Таблица теперь выглядит следующим образом:

Строка	***Номер счета***	***Баланс счета***	*ЗАБЛОКИРОВАНА*
1	123	(500,00 $) изменился и теперь равен 100,00 $	X
2	456	$240,25	
342,023	987	(100,00 $) изменился и теперь равен 500,00 $	X

Итак, обе измененные строки заблокированы для изменения. В этом отношении все СУБД работают примерно одинаково. Различия появятся, когда запрос SELECT доберется до заблокированных данных.

Добравшись до заблокированного блока, выполняемый запрос "поймет", что данные в нем изменились после начала выполнения. Чтобы дать согласованный (правильный) ответ, сервер Oracle в этот момент восстановит блок с заблокированными данными в том виде, как он был на момент начала выполнения запроса. Таким образом, сервер Oracle обходит блокировку — он читает, восстанавливая соответствующие данные из сегмента отката. Согласованный и правильный ответ получается без ожидания фиксации транзакции.

Однако СУБД, допускающая грязное чтение, просто вернет значение баланса счета 987 на момент его чтения, в данном случае 500 $. Запрос учтет перевод суммы 400 $ дважды и выдаст сумму, которая никогда не фигурировала в таблице счетов. В многопользовательской базе данных грязное чтение может быть опасным, и лично я никогда не видел от него пользы. В результате такого чтения не только выдается неверный результат, но могут выдаваться данные, никогда не существовавшие в базе данных. Пусть вместо перевода денег с другого счета транзакция добавила бы 400 $ на счет 987. Грязное чтение добавило бы 400 $ и выдало "правильный" ответ, не так ли? А если не зафиксированная транзакция будет отменена? Нам только что предъявили 400 $, которых в базе данных фактически никогда не было.

Дело в том, что грязное чтение — это не возможность, это просто лишняя ответственность. В Oracle брать ее на себя нет необходимости. Мы получаем все преимущества грязного чтения (отсутствие блокирования) без каких-либо некорректных результатов.

Уровень изолированности READ COMMITTED

Уровень изолированности READ COMMITTED требует, чтобы транзакция читала только данные, зафиксированные до ее начала. Никаких грязных чтений. Неповторяемость при чтении допускается (повторное чтение той же строки в транзакции может дать другой результат), как и чтение фантомов (по запросу могут возвращаться вновь вставленные строки, невидимые ранее для транзакции). READ COMMITTED, вероятно, — наиболее часто и повсеместно используемый в приложениях уровень изолированности транзакции. Другие уровни изолированности используются редко.

Уровень READ COMMITTED не настолько безобиден, как можно судить по названию. Если взглянуть на представленную ранее таблицу, все кажется вполне очевидным. Естественно, с учетом указанных выше правил, запрос к базе данных, выполненный в транзакции с уровнем изолированности READ COMMITTED, должен выполняться одинаково, не так ли? Нет. Если запрос возвращает несколько строк, практически в любой СУБД уровень изолированности READ COMMITTED может стать причиной появления некорректных данных в той же мере, что и грязное чтение.

В Oracle, благодаря использованию многовариантного доступа и согласованных по чтению запросов, результат запроса к таблице счетов при уровне изолированности READ COMMITTED будет таким же, как и в примере с READ UNCOMMITTED. Сервер Oracle восстанавливает измененные данные в том виде, какой они имели на момент начала выполнения запроса, возвращая ответ, соответствующий состоянию базы данных на тот же момент.

Теперь давайте рассмотрим, как представленный выше пример мог бы выполняться в режиме READ COMMITTED в других СУБД. Результат удивит вас. Повторим тот же пример:

- Мы находимся в середине таблицы, прочитав и просуммировав к этому моменту N строк.
- Другая транзакция перевела 400 $ со счета 123 на счет 987.
- Эта транзакция еще не зафиксирована, поэтому строки 123 и 987 заблокированы.

Мы знаем, как поведет себя СУБД Oracle, добравшись до счета 987: она прочитает измененные данные из сегмента отката, подсчитает баланс (значение окажется равным 100,00 $) и завершит запрос. Давайте рассмотрим, как другая СУБД, работающая в том же стандартном режиме READ COMMITTED, будет выдавать ответ.

Время	*Запрос*	*Транзакция по переводу со счета на счет*
T1	Читает строку 1, sum = 500 $.	
T2	Читает строку 2, sum == 740,25 $.	
T3		Изменяет строку 1, устанавливает исключительную блокировку на блок 1, **предотвращающую другие изменения и** чтение. В строке 1 теперь значение 100 $.
T4	Читает строку N, sum = ...	

Время	Запрос	Транзакция по переводу со счета на счет
T5		Изменяет строку 342023, устанавливает на нее исключительную блокировку. В строке теперь значение 500 $.
Тб	Читает строку 342023, обнаруживает, что она была изменена. Сеанс блокируется в ожидании доступа к соответствующей строке. Выполнение запроса приостанавливается.	
T7		Транзакция фиксируется.
T8	Читает строку 342023, находит в ней баланс 500 $ и выдает окончательный результат.	

Обратите внимание, что в другой СУБД запрос будет заблокирован, когда доберется до счета 987. Нашему сеансу придется ждать освобождения соответствующей строки, пока не зафиксируется транзакция, удерживающая исключительную блокировку. Это одна из причин появления у многих разработчиков плохой привычки фиксировать результаты в середине транзакции. В большинстве других СУБД изменения мешают чтению. Но самое печальное в этом то, что пользователь вынужден ожидать **неверного** результата. Как и в случае грязного чтения, получен результат, никогда не существовавший в базе данных, но в этом случае пришлось еще и ожидать.

Важный урок здесь в том, что разные СУБД, выполняющие запрос с одним, предположительно безопасным, уровнем изолированности транзакции, возвращают существенно отличающиеся результаты в абсолютно одинаковых обстоятельствах. Важно понимать, что в Oracle не блокирующее чтение дается не ценой некорректности результатов. Оказывается, иногда можно, как говорится, и рыбку съесть, и на елку влезть.

Уровень изолированности REPEATABLE READ

Целью включения **REPEATABLE READ** в стандарт SQL92 было обеспечение уровня изолированности транзакций, дающего согласованные, корректные результаты и предотвращающего потерю изменений. На двух примерах будет показано, как это достигается в Oracle и что происходит в других системах.

Получение согласованного ответа

При установке уровня изолированности **REPEATABLE READ** результаты запроса должны быть согласованными на определенный момент времени. Обычно СУБД (но не Oracle) достигают уровня **REPEATABLE READ** за счет установки строчных разделяемых блокировок чтения. Разделяемая блокировка чтения предотвращает изменение прочитанных данных другими сеансами. Это, несомненно, снижает параллелизм. В Oracle для получения согласованных по чтению результатов выбрана модель многовариантного доступа, обеспечивающая больший параллелизм.

При использовании многовариантного доступа в Oracle получается ответ, согласованный на момент начала выполнения запроса. В других СУБД при использовании разделяемых блокировок чтения получается результат, согласованный на момент завершения запроса — на момент, когда вообще можно получить результат (подробнее об этом — ниже).

В системе, использующей для обеспечения уровня изолированности **REPEATABLE READ** разделяемые блокировки чтения, строки будут блокироваться по мере их обработки запросом. Поэтому, возвращаясь к рассматривавшемуся выше примеру, по мере чтения запросом таблицы счетов, он будет устанавливать разделяемые блокировки чтения на каждой строке:

Время	*Запрос*	*Транзакция по переводу со счета на счет*
T1	Читает строку 1, sum = 500 $. На блоке 1 установлена разделяемая блокировка.	
T2	Читает строку 2, sum = 740,25 $. На блоке 2 установлена разделяемая блокировка.	
T3		Пытается изменить строку 1, но эта попытка блокируется. Транзакция приостанавливается, пока не сможет установить исключительную блокировку.
T4	Читает строку N, sum = ...	
T5		Читает строку 342023, находит баланс счета, 100 $, и дает окончательный результат.
T6	Фиксирует транзакцию.	
T7		Изменяет строку 1, устанавливая на соответствующий блок исключительную блокировку. Теперь баланс в этой строке имеет значение 100 $.
T8		Изменят строку 342023, устанавливая на соответствующий блок исключительную блокировку. Теперь баланс в этой строке имеет значение 500 $. Транзакция фиксируется.

Эта таблица показывает, что корректный результат получен благодаря физическому упорядочению двух транзакций. Вот один из побочных эффектов использования разделяемых блокировок чтения для получения согласованных результатов: **сеансы считывания данных блокируют сеансы записи данных.** Кроме того, в этих системах сеансы записи данных блокируют сеансы их считывания.

Итак, понятно, как разделяемые блокировки чтения снижают параллелизм, но они также могут приводить к возникновению спорадических ошибок. В следующем примере мы начнем с исходной таблицы счетов, но в этот раз целью будет перевод 50,00 $ со счета 987 на счет 123.

Время	*Запрос*	*Транзакция по переводу со счета на счет*
T1	Читает строку 1, sum = 500 $. На строке 1 установлена разделяемая блокировка.	
T2	Читает строку 2, sum = 740,25 $. На строке 2 установлена разделяемая блокировка.	
T3		Изменяет строку 342023, устанавливает исключительную блокировку на строку 342023, предотвращающую другие изменения и установку разделяемых блокировок. В этой строке теперь содержится значение 50 $.
T4	Читает строку N, sum = ...	
T5		Пытается изменить строку 1, но она заблокирована. Транзакция приостанавливается до тех пор, пока не появится возможность установить исключительную блокировку.
T6	Пытается прочитать строку 342023, но не может, поскольку на нее уже установлена исключительная блокировка.	

Только что мы получили классическую ситуацию взаимной блокировки. Наш запрос удерживает ресурс, необходимый транзакции, изменяющей данные, и наоборот. Запрос и изменяющая транзакция взаимно заблокировали друг друга. Один из сеансов будет выбран в качестве жертвы, и его транзакция будет прекращена. Затрачено немало времени и ресурсов и все впустую: произошел откат. Это — второй побочный эффект разделяемых блокировок чтения: **сеансы чтения и сеансы записи данных могут взаимно блокировать друг друга, и часто так и происходит.**

Как уже было показано, в Oracle обеспечивается согласованность по чтению на уровне операторов без блокирования сеансов записи сеансами чтения или взаимного блокирования. Сервер Oracle **никогда** не использует разделяемые блокировки чтения. Разработчики Oracle выбрали более сложную в реализации, но обеспечивающую принципиально более высокую степень параллелизма схему многовариантного доступа.

Предотвращение потери изменений

Чаще всего уровень изолированности транзакции **REPEATABLE READ** используется для предотвращения потери изменений. При установке уровня **REPEATABLE READ** это не происходит. По определению (повторяющиеся чтения) повторное чтение строки в той же транзакции даст точно такой же результат.

В других СУБД, кроме Oracle, **REPEATABLE READ** может реализовываться с помощью **SELECT FOR UPDATE** и разделяемых блокировок чтения. Если два пользователя выбрали одну и ту же строку для изменения, они оба устанавливают на нее разделяе-

мую блокировку чтения. Когда первый пользователь попытается изменить строку, эта попытка будет заблокирована. В случае попытки второго пользователя выполнить изменение возникнет взаимная блокировка. Это не идеально, но предотвращает потерю изменений.

Если в Oracle необходим уровень изолированности **REPEATABLE READ,** но не хочется физически упорядочивать доступ к таблице с помощью операторов **SELECT FOR UPDATE NOWAIT** (как было продемонстрировано в начале главы), придется устанавливать уровень изолированности **SERIALIZABLE. SERIALIZABLE** покрывает все более либеральные уровни изолированности, поэтому, если можно выполнять доступ уровня **SERIALIZABLE,** то возможен доступ и уровня **REPEATABLE READ.**

В Oracle транзакция с уровнем изолированности **SERIALIZABLE** реализуется так, что согласованность по чтению, обычно получаемая на уровне оператора, распространяется на всю транзакцию. То есть результаты каждого выполняемого в транзакции запроса соответствуют состоянию базы данных на момент начала транзакции. Если выполнить в этом режиме:

```
Select * from T;
Begin dbms_lock.sleep(60*60*24); end;
Select * from T;
```

результаты, возвращаемые из таблицы T, будут одинаковыми и через 24 часа (или мы получим сообщение об ошибке **ORA-1555, snapshot too old).** Уровень изолированности гарантирует, что эти два запроса всегда будут возвращать одинаковые результаты. В Oracle это достигается тем же способом, что и согласованность по чтению одного запроса. Сервер использует сегменты отката для воссоздания данных в том виде, как они были на момент начала транзакции, а не на момент начала выполнения оператора. Если же мы пытаемся изменить данные в транзакции с уровнем изолированности **SERIALIZABLE** и обнаруживаем, что данные изменились после ее начала, мы получаем сообщение об ошибке, информирующее о том, что невозможно обеспечить последовательность доступа. Вскоре мы рассмотрим это подробнее.

Понятно, что этот подход не оптимален в случае рассмотренного в начале главы приложения для отдела кадров. В этом приложении типична ситуация, когда оба пользователя запрашивают данные, а затем оба изменяют их на экране. Первый пользователь пытается сохранить изменения, и у него это успешно получается. Второй же пользователь при попытке сохранить изменения получит сообщение об ошибке. Он зря потратил время. Ему придется перезапустить транзакцию, получить произошедшие за это время изменения и сделать все сначала. Потеря изменений предотвращается, но ценой дополнительных неудобств для пользователя. Однако если ситуация требует использования уровня изолированности транзакции **REPEATABLE READ и** не предполагается одновременное изменение несколькими транзакциями одних и тех же строк, то использование уровня изолированности **SERIALIZABLE** вполне допустимо.

Уровень изолированности SERIALIZABLE

Этот уровень изолированности транзакции обычно считают наиболее ограничивающим, но он обеспечивает самую высокую степень изолированности. Транзакция с уровнем изолированности **SERIALIZABLE** работает в среде, где как бы нет других пользо-

вателей, изменяющих данные в базе данных; база данных "замораживается" на момент начала транзакции. Для транзакции база данных выглядит согласованной, в виде своего рода моментального снимка. Побочные эффекты (изменения) от других транзакций для такой транзакции невидимы, независимо от того, как долго она выполняется. Уровень изолированности **SERIALIZABLE** не **означает,** что все пользовательские транзакции дают такой же результат, как и при последовательном выполнении. Он не предполагает наличия физического порядка выполнения транзакций, дающего такой же результат. Это последнее утверждение часто понимают неправильно, и небольшой пример сейчас все прояснит. В следующей таблице представлены два сеанса, выполняющие определенную последовательность действий. Таблицы базы данных A и B первоначально пусты и созданы следующими операторами:

```
tkyte@TKYTE816> create table a (x int) ;
Table created.

tkyte@TKYTE816> create table b (x int) ;
Table created.
```

Теперь происходят следующие события:

Время	*Сеанс 1 выполняет*	*Сеанс 2 выполняет*
0:00	Alter session set isolation_level=serializable;	
0:01		Alter session set isolation_level=serializable;
0:02	Insert into a select count(*) from b;	
0:03		Insert into b select count(*) from a;
0:04	Commit;	
0:05		Commit;

Теперь в каждой из таблиц имеется по строке со значением ноль. Если бы выполнялось "последовательное" упорядочение транзакций, мы не могли бы получить нулевые значения в обеих таблицах. Если бы сеанс 1 опережал сеанс 2, то в таблице B было бы значение 1. Если бы сеанс 2 выполнялся прежде сеанса 1, то в таблице A было бы значение I. Однако при выполнении приведенной выше последовательности действий в обеих таблицах будут **нулевые** значения, т.е. транзакции выполняются так, будто других транзакций в базе данных в этот момент нет. Неважно, сколько раз сеанс 1 запрашивает таблицу B, — будет выдано количество строк, зафиксированных в базе данных на момент времени 0:00. Аналогично, независимо от количества запросов таблицы A, в сеансе 2 будет получено значение на момент времени 0:01.

В Oracle последовательность достигается путем распространения согласованности по чтению, получаемой на уровне оператора, на уровень транзакции. Результаты не согласуются на момент начала выполнения оператора — они предопределены моментом начала транзакции. Весьма глубокая мысль: базе данных известен ответ на любой возможный ваш вопрос прежде, чем вы его зададите.

Этот уровень изолированности не дается даром: за него нужно платить сообщением об ошибке:

```
ERROR at line 1:
ORA-08177: can't serialize access for this transaction
```

Это сообщение будет получено при любой попытке обновления строки, измененной после начала транзакции. Сервер Oracle придерживается при обеспечении последовательности оптимистического подхода; он предполагает, что данные, которые потребуется изменять вашей транзакции, не будут изменены другой транзакцией. Обычно именно так и происходит, и подобное предположение вполне оправдано, особенно в системах оперативной обработки транзакций (ООТ). Если в ходе транзакции ваши данные не изменяются другим сеансом, этот уровень изолированности, обычно снижающий степень параллелизма в других системах, обеспечит ту же степень параллелизма, что и при отсутствии транзакций с уровнем изолированности **SERIALIZABLE.** Вопрос лишь в том, не получите ли вы сообщения об ошибке **ORA-08177,** если предположение не оправдается. Однако поразмыслив, можно найти такой риск оправданным. Если уж используется транзакция с уровнем изолированности **SERIALIZABLE,** нет смысла ожидать изменения той же информации другими транзакциями. Если это возможно, используйте оператор **SELECT ... FOR UPDATE,** как было показано выше, который обеспечит требуемую последовательность доступа. Итак, если

- высока вероятность того, что данные не изменяет другой сеанс;
- необходима согласованность по чтению на уровне транзакций;
- транзакции будут непродолжительными (чтобы первое условие стало более реальным);

то использование уровня изолированности **SERIALIZABLE** дает хороший эффект. Корпорация Oracle считает этот метод настолько масштабируемым, что выполняет на таком уровне изолированности все тесты **TPC-C** (стандартный набор тестов производительности систем ООТ; подробнее см. на сайте **http://www.tpc.org).** Во многих других реализациях это достигается путем использования разделяемых блокировок чтения со всеми соответствующими взаимными блокировками и ожиданием. В Oracle блокирования нет, но, если другие сеансы изменяют данные, необходимые вашему сеансу, будет получено сообщение об ошибке **ORA-08177.** Однако это сообщение об ошибке генерируется гораздо реже, чем происходят взаимные блокировки и ожидания снятия блокировок в других системах.

Транзакции только для чтения

Транзакции только для чтения очень похожи на транзакции с уровнем изолированности **SERIALIZABLE.** Единственное отличие в том, что они не разрешают изменять данные, поэтому не подвержены ошибке **ORA-08177.** Транзакции только для чтения предназначены для создания отчетов, когда данные отчета должны быть согласованы по отношению к определенному моменту времени. В других системах придется использовать уровень изолированности **REPEATABLE READ,** получая нежелательные последствия разделяемых блокировок чтения. В Oracle используется транзакция только для

чтения. В этом режиме результат, получаемый в отчете на основе данных, собранных 50 операторами SELECT, будет согласован по отношению к одному моменту — началу транзакции. Это можно сделать, не блокируя ни одного компонента данных, где бы то ни было.

Достигается это с помощью того же многовариантного доступа, что и для отдельных операторов. Данные восстанавливаются при необходимости из сегментов отката и представляются в том виде, какой они имели до начала создания отчета. Однако использование транзакций только для чтения также не лишено проблем. Тогда как сообщение об ошибке **ORA-08177** вполне вероятно для транзакций с уровнем изолированности SERIALIZABLE, при использовании транзакций только для чтения можно ожидать сообщений об ошибке **ORA-1555 snapshot too old.** Они будут выдаваться в системе, где другие сеансы активно изменяют считываемую нами информацию. Их изменения *(данные отмены* — undo) записываются в сегменты отката. Но сегменты отката используются циклически, аналогично журналам повторного выполнения. Чем более продолжительное время создается отчет, тем выше вероятность, что часть данных отмены, необходимых для восстановления данных, перестанет быть доступной. Сегмент отката будет использоваться повторно, и необходимая нам его часть окажется использованной другой транзакцией. В этот момент сеанс и получит сообщение об ошибке **ORA-1555,** и придется начинать все с начала. Единственное решение этой болезненной проблемы — сконфигурировать в системе сегменты отката адекватного размера. Постоянно приходится сталкиваться с ситуацией, когда, пытаясь сэкономить несколько мегабайт дискового пространства, создают сегменты отката минимально возможного размера (зачем "тратить" место на то, что мне фактически не нужно?). Проблема в том, что сегменты отката — ключевой компонент, обеспечивающий функционирование базы данных, и если их размер окажется меньше, чем нужно, — ошибка **ORA-1555** гарантирована. За 12 лет использования Oracle 6, 7 и 8 я никогда не сталкивался с ошибкой **ORA-1555** за пределами тестовой системы или среды разработки. При возникновении этих ошибок сразу становится понятно, что сегменты отката имеют недостаточный размер, но это легко исправляется. Мы еще вернемся к этой проблеме в главе 5.

Резюме

В этой главе мы рассмотрели большой объем материала, иногда трудного для понимания. Тогда как блокирование — тема весьма простая, некоторые его побочные эффекты понять сложнее. Однако знание всего этого жизненно важно. Например, если не знать о блокировке таблицы, используемой сервером Oracle для обеспечения требования внешнего ключа, когда он не проиндексирован, приложения будут иметь низкую производительность. Если не представлять себе, как определить по словарю данных, кто кого блокирует, трудно разобраться в причине происходящего. Просто покажется, что временами база данных "зависает". Если бы я получал по доллару всякий раз, когда устранял "неразрешимую" проблему зависания с помощью запроса, выявляющего неиндексированные внешние ключи и предлагающего соответствующий индекс, я бы уже был очень богатым человеком.

Мы выяснили смысл уровней изолированности транзакции, устанавливаемых стандартом SQL92, и то, как эти уровни реализуются в Oracle и в других СУБД. Мы убедились, что в тех случаях, когда для обеспечения согласованности данных используются блокировки чтения, приходится идти на компромисс между параллелизмом и согласованностью данных. Для получения высокой степени параллелизма при доступе к данным необходимо снизить требования к согласованности результатов. Для получения согласованных, правильных результатов необходимо снижать параллелизм. Мы увидели, что в случае сервера Oracle, благодаря многовариантному доступу, такой проблемы не существует. Приведенная ниже небольшая таблица наглядно демонстрирует, чего можно ожидать от СУБД, использующей блокирование при чтении, по сравнению с многовариантным доступом в Oracle:

Уровень изолированности	*Реализация*	*Запись блокирует чтение*	*Чтение блокирует запись*	*Чтения могут привести к взаимным блокировкам*	*Некорректные результаты запроса*	*Потерянные изменения*	*Эскалация или ограничения количества блокировок*
READ UNCOMMITTED	Блокирование при чтении	Нет	Нет	Нет	Да	Да	Да
READ COMMITTED	(Другие СУБД)	Да	Нет	Нет	Да	Да	Да
REPEATABLE READ		Да	Да	Да	Нет	Нет	Да
SERIALIZABLE		Да	Да	Да	Нет	Нет	Да
READ COMMITTED	Многовариантный доступ	Нет	Нет	Нет	Нет	Нет*	Нет
SERIALIZABLE	(Oracle)	Нет	Нет	Нет	Нет	Нет	Нет

* *При использовании select for update nowait*

Все, что касается средств управления одновременным доступом и их реализации в СУБД, необходимо глубоко усвоить. Я пел дифирамбы многовариантному доступу и согласованности по чтению, но, как и все под небесами, эти средства неоднозначны. Если не понимать, что они собой представляют и как работают, ошибки при разработке приложений неизбежны. Рассмотрим пример с планированием ресурсов из главы 1. В СУБД без многовариантного доступа и связанных с ним не блокируемых чтений подход, изначально заложенный в программу, может себя вполне оправдать. Однако этот подход неприемлем в СУБД Oracle — будет нарушаться целостность данных. Если не знать возможных последствий, можно написать программу, повреждающую данные. Вот так — все просто.

4

Транзакции

Транзакции — одно из свойств, отличающих базу данных от файловой системы. В файловой системе, если сбой ОС происходит во время записи файла, он, скорее всего, окажется поврежденным. Существуют различные "журнализируемые" файловые системы, которые позволяют восстановить файл на определенный момент времени. Однако, если необходима синхронизация двух файлов, это не поможет: когда при изменении одного файла сбой происходит до того как завершится изменение второго, файлы окажутся рассинхронизированными.

Основное назначение транзакций в базе данных — переводить ее из одного согласованного состояния в другое. При фиксации изменений в базе данных гарантируется сохранение либо всех изменений, либо ни одного. Более того, выполняются все правила и проверки, обеспечивающие целостность данных.

Транзакции базы данных обладают свойствами, сокращенно называемыми **ACID** (Atomicity, Consistency, Isolation, Durability). Вот что означают эти свойства:

- **Неделимость** (Atomicity). Транзакция либо выполняется полностью, либо не выполняется.
- **Согласованность** (Consistency). Транзакция переводит базу данных из одного согласованного состояния в другое.
- **Изолированность** (Isolation). Результаты транзакции становятся доступны для других транзакций только после ее фиксации.
- **Продолжительность** (Durability). После фиксации транзакции изменения становятся постоянными.

Транзакции в Oracle обладают всеми перечисленными выше характеристиками. В этой главе мы опишем влияние неделимости на выполнение операторов в СУБД Oracle. Будут рассмотрены операторы управления транзакцией, такие как **COMMIT, SAVEPOINT** и **ROLLBACK,** и то, как в транзакции обеспечивается выполнение требований целостности. Мы также постараемся выяснить, почему приобретаются плохие привычки в работе с транзакциями при разработке приложений для других СУБД. Разберемся с распределенными транзакциями и *двухэтапной фиксацией.* Наконец, рассмотрим реальные проблемы, возникающие при использовании и журнализации транзакций, а также выясним, какую роль могут играть сегменты отката.

Операторы управления транзакцией

В СУБД Oracle нет оператора "начать транзакцию". Транзакция неявно начинается с первого же оператора, изменяющего данные (установившего блокировку TX). Операторы **COMMIT** или **ROLLBACK** явно завершают транзакции. Всегда завершайте транзакции явно с помощью оператора **COMMIT** или **ROLLBACK,** иначе решение о том, фиксировать или откатывать, автоматически примет используемое инструментальное средство или среда. При обычном выходе из сеанса SQL*Plus без фиксации или отката эта утилита предполагает, что нужна фиксация, и автоматически ее выполняет. При завершении же программы на языке **Pro*C** по умолчанию выполняется откат.

Транзакции в Oracle неделимы: либо фиксируется (делается постоянным) результат выполнения каждого из операторов, составляющих транзакцию, либо результаты выполнения всех операторов откатываются. Эта защита распространяется и на отдельные операторы. Оператор либо завершается полностью успешно, либо полностью откатывается. Обратите внимание: я написал, что **оператор** откатывается. Сбой в одном операторе не вызывает автоматического отката ранее выполненных операторов. Их результаты сохраняются и должны быть зафиксированы или отменены пользователем. Прежде чем разбираться детально, что означает свойство "неделимости" для оператора и транзакции, рассмотрим операторы управления транзакциями.

- **COMMIT.** В простейшем случае достаточно ввести просто **COMMIT.** Можно быть более многословным и выполнить **COMMIT WORK,** но обе формы эквивалентны. Оператор **COMMIT** завершает транзакцию и делает любые выполненные в ней изменения постоянными (продолжительными). В распределенных транзакциях используются расширения оператора **COMMIT.** Эти расширения позволяют пометить оператор **COMMIT** (точнее, пометить транзакцию), задав для него комментарий, а также принудительно зафиксировать сомнительную распределенную транзакцию.
- **ROLLBACK. В** простейшем случае выполняется просто оператор **ROLLBACK.** Можно также использовать форму **ROLLBACK WORK,** но обе формы эквивалентны. Простой оператор отката завершает транзакцию и отменяет все выполненные в ней и незафиксированные изменения. Для этого он читает информацию из сегментов отката и восстанавливает блоки данных в состояние, в котором они находились до начала транзакции.

- SAVEPOINT. Оператор SAVEPOINT позволяет создать в транзакции "метку", или точку сохранения. В одной транзакции можно выполнять оператор SAVEPOINT несколько раз, устанавливая несколько точек сохранения.
- **ROLLBACK TO <точка сохранения>**. Этот оператор используется совместно с представленным выше оператором **SAVEPOINT.** Транзакцию можно откатить до указанной точки сохранения, не отменяя все сделанные до нее изменения. Таким образом, можно выполнить два оператора **UPDATE,** затем — оператор **SAVEPOINT,** а после него — два оператора **DELETE.** При возникновении ошибки или исключительной ситуации в ходе выполнения операторов **DELETE** транзакция будет откатываться до указанной оператором **SAVEPOINT** точки сохранения; при этом будут отменяться операторы **DELETE,** но не операторы **UPDATE.**
- **SET TRANSACTION.** Этот оператор позволяет устанавливать атрибуты транзакции, такие как уровень изолированности и то, будет ли она использоваться только для чтения данных или для чтения и записи. Этот оператор также позволяет привязать транзакцию к определенному сегменту отката.

Вот и все операторы управления транзакциями. Чаще всего используются операторы **COMMIT** и **ROLLBACK.** Оператор **SAVEPOINT** имеет несколько специфическое назначение. Сервер Oracle часто использует его по ходу работы, но определенную пользу от него можно получить и в приложениях.

Теперь можно приступить к изучению последствий неделимости оператора и транзакции. Рассмотрим следующий оператор:

```
Insert into t values (1);
```

Кажется вполне очевидным, что строка вставлена не будет, если из-за нарушения требования уникальности этот оператор не выполнится. Рассмотрим, однако, следующий пример, где при вставке или удалении строки в таблице T срабатывает триггер, изменяющий значение столбца **cnt** в таблице T2:

```
tkyte@TKYTE816> create table t2 (cnt int);
Table created.

tkyte@TKYTE816> insert into t2 values (0);
1 row created.

tkyte@TKYTE816> create table t (x int check (x>0));
Table created.

tkyte@TKYTE816> create trigger t_trigger
  2 before insert or delete on t for each row
  3 begin
  4    if (inserting) then
  5      update t2 set cnt = cnt +1;
  6    else
  7      update t2 set cnt = cnt -1;
  8    end if;
  9    dbms_output.put_line('I fired and updated ' ||
                                    sql%rowcount || ' rows');
```

```
 10  end;
 11  /
Trigger created.
```

В этой ситуации предполагаемые последствия менее очевидны. Если ошибка происходит после срабатывания триггера, должны ли остаться в силе изменения, сделанные триггером? То есть, если триггер сработал, изменив таблицу T2, а строка не вставлена в таблицу T, каким должен быть результат? Естественно, не хотелось бы, чтобы значение в столбце **cnt** таблицы T2 было увеличено, если строка в таблицу T не вставлена. К счастью, в Oracle переданный клиентом оператор **(INSERT INTO T** в данном случае) либо выполняется успешно полностью, либо не выполняется вообще. Это — неделимый оператор. В этом можно убедиться следующим образом:

```
tkyte@TKYTE816> set serveroutput on

tkyte@TKYTE816> insert into t values (1);
I fired and updated 1 rows

1 row created.

tkyte@TKYTE816> insert into t values (-1);
insert into t values (-1)
*
ERROR at line 1:
ORA-02290: check constraint (TKYTE.SYS_C001570) violated

tkyte@TKYTE816> exec null /* это необходимо для получения    */
                          /* результатов dbms_output         */
I fired and updated 1 rows

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select * from t2;
     CNT

       1
```

Мы успешно вставили одну строку в таблицу T, получив надлежащее сообщение: **I fired and updated 1 rows.** Следующий оператор **INSERT** нарушает ограничение целостности таблицы T. Мне пришлось ввести **exec NULL** и выполнить тем самым пустой оператор, чтобы утилита SQL*Plus показала информацию, выданную с помощью пакета **DBMS_OUTPUT,** (поскольку SQL*Plus не выдает содержимое буфера DBMS_OUTPUT после выполнения оператора SELECT), но и в этот раз было получено сообщение, что триггер сработал и изменил одну строку. Можно было бы предположить, что в таблице T2 теперь будет значение 2, но мы видим там значение I. Сервер Oracle обеспечил неделимость переданного оператора вставки.

Он достигает этого, неявно размещая операторы **SAVEPOINT** вокруг каждого из переданных операторов. Представленные выше операторы вставки на самом деле обрабатываются так:

```
Savepoint statement1;
    Insert into t values (1) ;
If error then rollback to statement1;
Savepoint statement2;
    Insert into t values (-1) ;
If error then rollback to statement2;
```

Для программистов, привыкших работать с СУБД Sybase или SQL Server, это поначалу кажется странным. В этих СУБД все происходит **в точности наоборот. В** этих системах триггеры выполняются независимо от вызвавшего их срабатывание оператора. Если в них происходит ошибка, триггеры должны явно откатывать сделанные ими изменения, а затем возбуждать другую исключительную ситуацию для отката оператора, вызвавшего срабатывание. В противном случае выполненные триггером изменения могут остаться в силе, даже если вызвавший срабатывание оператор завершился неудачно.

В Oracle неделимость операторов распространяется на любую необходимую глубину. В рассмотренном выше примере оператор **INSERT INTO** T вызвал срабатывание триггера, изменившего другую таблицу, а для этой таблицы есть триггер, удаляющий строки из еще одной таблицы, и так далее, но в любом случае либо **все** изменения выполняются успешно, либо ни **одно.** Для этого не надо применять специальные приемы кодирования — все работает именно так.

Интересно отметить, что сервер Oracle считает оператором анонимный блок PL/SQL. Рассмотрим следующую хранимую процедуру:

```
tkyte@TKYTE816> create or replace procedure p
  2  as
  3  begin
  4            insert into t values (1) ;
  5            insert into t values (-1) ;
  6  end;
  7  /
Procedure created.

tkyte@TKYTE816> select * from t;
no rows selected

tkyte@TKYTE816> select * from t2;

     CNT
```

Итак, имеется процедура, которая не должна сработать. Второй оператор вставки всегда будет завершаться неудачно. Давайте посмотрим, что произойдет, если выполнить эту хранимую процедуру:

```
tkyte@TKYTE816> begin
  2     p;
  3  end;
  4  /
I fired and updated 1 rows
I fired and updated 1 rows
begin
```

```
ERROR at line 1:
ORA-02290: check constraint (TKYTE.SYS_C001570) violated
ORA-06512: at "TKYTE.P", line 5
ORA-06512: at line 2

tkyte@TKYTE816> select * from t;

no rows selected

tkyte@TKYTE816> select * from t2;

     CNT

       0
```

Как видите, сервер Oracle счел вызов процедуры неделимым оператором. Клиент послал блок кода **(BEGIN P; END;),** и сервер Oracle окружил его операторами **SAVEPOINT.** Поскольку процедура P не сработала, сервер Oracle восстановил базу данных в состояние, предшествовавшее ее вызову. Теперь, послав немного измененный блок, мы получим абсолютно другой результат:

```
tkyte@TKYTE816> begin
  2            P;
  3  exception
  4           when others then null;
  5  end;
  6  /
I  fired and updated 1 rows
I  fired and updated 1 rows

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select * from t;
      X

      1

tkyte@TKYTE816> select * from t2;

    CNT

      1
```

Мы выполнили блок кода, в котором игнорируются любые ошибки, и результат получился принципиально другим. Тогда как первый вызов процедуры P не вызвал никаких изменений, в данном случае первый оператор **INSERT** выполняется успешно, и столбец **cnt** в таблице T2 увеличивается на 1. Сервер Oracle считает "оператором" переданный клиентом блок. Этот оператор завершается успешно, самостоятельно перехватывая и игнорируя ошибку, так что фрагмент **"If error then rollback..."** не срабатывает, и сервер Oracle не выполняет откат до точки сохранения после его выполнения. Вот по-

чему результаты работы процедуры частично P сохраняются. Причина этого частичного сохранения прежде всего в том, что внутри P соблюдается неделимость операторов: все операторы в процедуре P неделимы. Процедура P работает как клиент Oracle при посылке двух операторов INSERT. Каждый оператор INSERT либо успешно выполняется полностью, либо завершается неудачно. Это подтверждается тем фактом, что триггер таблицы T срабатывает дважды и два раза изменяет таблицу T2, хотя счетчик в T2 отражает только одно изменение. Вокруг второго оператора INSERT, выполненного в процедуре P, были установлены неявные точки сохранения.

Различие между двумя приведенными блоками кода невелико, но его надо учитывать при разработке приложений. Добавление обработчика исключительных ситуаций в блок кода PL/SQL может радикально изменить его работу. Ниже приведен более корректный способ реализации того же подхода, расширяющий неделимость на уровне оператора до уровня всего блока PL/SQL:

```
tkyte@TKYTE816> begin
  2       savepoint sp;
  3       P;
  4  exception
  5       when others then
  6            rollback to sp;
  7  end;
  8  /
I fired and updated 1 rows
I fired and updated 1 rows

PL/SQL procedure successfully completed.

tkyte@TKYTE816>

tkyte@TKYTE816> select * from t;

no rows selected

tkyte@TKYTE816> select * from t2;

     CNT

       0
```

Имитируя автоматически выполняемые сервером Oracle действия с помощью оператора SAVEPOINT, мы можем восстановить исходное поведение процедуры и при этом перехватывать и "игнорировать" ошибки.

Требования целостности и транзакции

Интересно разобраться, когда именно проверяются требования целостности. По умолчанию требования целостности проверяются после выполнения всего SQL-оператора. Обратите внимание: я написал "SQL-оператора", а не просто "оператора". Если в хранимой процедуре PL/SQL есть несколько операторов SQL, проверка требований целостности каждого оператора выполняется сразу же по завершении его выполнения, а не

по завершении выполнения всей хранимой процедуры. Проверка требований целостности может быть программно отложена до завершения транзакции или до момента, когда разработчик сочтет необходимым их проверить.

Итак, почему же требования проверяются после выполнения SQL-оператора, а не при выполнении? А потому, что одному оператору "позволено" кратковременно делать отдельные строки таблицы несогласованными. Попытка получить частичные результаты выполнения оператора приведет к отмене этих результатов сервером Oracle, даже если конечные результаты вполне допустимы. Пусть, например, имеется следующая таблица:

```
tkyte@TKYTE816> create table t (x int unique);
Table created.

tkyte@TKYTE816> insert into t values (1);
1 row created.

tkyte@TKYTE816> insert into t values (2) ;
1 row created.
```

Теперь, пусть необходимо выполнить изменение нескольких строк:

```
tkyte@TKYTE816> update t set x = x+1;
2 rows updated.
```

Если бы сервер Oracle проверял требование после изменения каждой строки, то вероятность неудачного завершения изменений составляла бы 50%. Строки в таблице T просматриваются в определенном порядке; поэтому, если бы сначала изменялась строка с X=1, значение моментально дублировалось бы в столбце X, а изменение отвергалось. Поскольку сервер Oracle терпеливо ждет завершения оператора, он выполняется успешно, так как к моменту его завершения дублирующихся значений нет.

Версии, начиная с Oracle 8.0, позволяют **отложить** проверку требований. Эта возможность может оказаться крайне полезной при выполнении различных операций. Например, в случае каскадного распространения изменения первичного ключа на внешние ключи. Многие скажут, что этого никогда не придется делать, что первичные ключи должны быть неизменны (и я тоже так скажу), но часто пользователи настаивают на возможности каскадного изменения. Отложенная проверка изменений позволяет это сделать.

В предыдущих версиях в общем-то тоже можно было сделать каскадное изменение, но для этого приходилось выполнять огромный объем работы, да и сам подход имел определенные ограничения. С помощью отложенной проверки изменений задача становится практически тривиальной. Решение выглядит примерно так:

```
tkyte@TKYTE816> create table p
  2   (pk  int primary key)
  3   /
Table created.

tkyte@TKYTE816>
tkyte@TKYTE816> create table c
```

```
  2   (fk   int constraint c_fk
  3              references p(pk)
  4              deferrable
  5              initially immediate
  6   )
  7   /
Table created.

tkyte@TKYTE816> insert into p values (1);
1 row created.

tkyte@TKYTE816> insert into c values (1);
1 row created.
```

Итак, имеется родительская таблица P и подчиненная таблица C. Таблица C ссылается на таблицу P, а реализующее это правило требование названо **C_FK** (child foreign key — внешний ключ подчиненной таблицы). Это требование создано как допускающее отложенную проверку — **DEFERRABLE,** но при создании, кроме того, указана конструкция **INITIALLY IMMEDIATE.** То есть проверку требования можно отложить до фиксации транзакции или любого другого момента. По умолчанию, однако, проверка будет выполняться на уровне оператора. Именно так чаще всего и используются требования с отложенной проверкой. Существующие приложения не будут проверять нарушение требования перед выполнением оператора **COMMIT,** поэтому лучше не делать им таких сюрпризов. В соответствии с определением, таблица C будет вести себя обычным образом, но позволит явно это поведение изменить. Теперь давайте попытаемся применить ряд операторов ЯМД к таблицам и посмотрим, что получится:

```
tkyte@TKYTE816> update p set pk = 2;
update p set pk = 2
*
ERROR at line 1:
ORA-02292: integrity constraint (TKYTE.C_FK) violated - child record found
```

Поскольку требование сейчас работает в режиме **IMMEDIATE,** этот изменение не будет выполнено. Изменив режим требования, попробуем снова:

```
tkyte@TKYTE816> set constraint c_fk deferred;
Constraint set.

tkyte@TKYTE816> update p set pk = 2;
1 row updated.
```

Теперь все получилось. В демонстрационных целях я собираюсь показать, как, процедурно проверив требование в режиме **DEFERRED,** убедиться, что сделанные изменения согласуются с установленными бизнес-правилами (другими словами, не нарушено ли требование). Имеет смысл делать это перед фиксацией или передачей управления в другую часть программы (где ограничений в режиме **DEFERRED** может и не быть):

```
tkyte@TKYTE816> set constraint c_fk immediate;
set constraint c fk immediate
```

```
ERROR at line 1:
ORA-02291: integrity constraint (TKYTE.C_FK) violated – parent key not found
```

Этот оператор не выполняется и сразу же возвращает сообщение об ошибке, чего и следовало ожидать: мы ведь знаем, что требование нарушено. Изменение таблицы P при этом не отменяется (это нарушило бы неделимость на уровне оператора). Учтите также, что транзакция по-прежнему работает с требованием **C_FK** в режиме **DEFERRED,** поскольку оператор **SET CONSTRAINT** не сработал. Продолжим, распространяя изменение на таблицу C:

```
tkyte@TKYTE816> update c set fk = 2;
1 row updated.

tkyte@TKYTE816> set constraint c_fk immediate;
Constraint set.

tkyte@TKYTE816> commit;
Commit complete.
```

Именно так это и происходит.

Плохие привычки при работе с транзакциями

У многих разработчиков выработались плохие привычки в отношении транзакций. Особенно часто это наблюдается у разработчиков, которые имели дело с СУБД, поддерживающими транзакции, но не навязывающими их использование. Например, в СУБД Informix (по умолчанию), Sybase и SQL Server необходимо явно начинать транзакцию. В противном случае каждый оператор будет выполняться как отдельная транзакция. Как сервер Oracle неявно устанавливает вокруг отдельных операторов **SAVEPOINT,** в этих СУБД вокруг каждого оператора неявно устанавливаются операторы **BEGIN WORK/COMMIT** или **ROLLBACK.** Это объясняется тем, что в упомянутых СУБД блокировки — ценный ресурс, а сеансы, читающие данные, блокируют сеансы, изменяющие данные. Пытаясь повысить параллелизм, создатели этих СУБД заставляют разработчиков делать транзакции как можно короче (иногда в ущерб целостности данных).

В СУБД Oracle принят другой подход. Транзакции всегда начинаются неявно, и возможность автоматической фиксации отсутствует, если только она не реализована в приложении (см. описание функционального интерфейса JDBC в конце этого раздела). В Oracle транзакция фиксируется, когда это необходимо, и не раньше. Транзакции должны быть такого размера, как нужно для приложения. Проблемы нехватки блокировок, невозможности доступа к данным для других сеансов и т.п. учитывать не надо — размер транзакции определяется исключительно требованиями обеспечения целостности данных. Блокировки не являются особо ценным ресурсом, а одновременное чтение и запись данных не приводит к конфликтам. Это обеспечивает надежность транзакций в базе данных. Продолжительность транзакций не ограничивается, — она определяется требованиями приложения. Транзакции не должны быть ориентированы на удобство работы

компьютера и его программного обеспечения — они предназначены для защиты целостности данных пользователей.

Столкнувшись с задачей изменения большого количества строк, программисты обычно пытаются изобрести процедурный способ сделать это в цикле — так, чтобы можно было фиксировать изменения группы строк определенного размера. Я слышал два обоснования подобного решения:

- быстрее и эффективнее фиксировать изменения часто с помощью множества небольших транзакций, чем обработать все строки и зафиксировать одну большую транзакцию;
- недостаточно места в сегментах отката.

Оба эти утверждения неверны. Частые фиксации изменений не способствуют ускорению — почти всегда быстрее сделать все необходимое в одном SQL-операторе. Рассмотрим небольшой пример: имеется, например, таблица T с множеством строк, и необходимо изменить значение в столбце для каждой строки этой таблицы. Это можно сделать одним простым оператором:

```
tkyte@TKYTE816> create table t as select * from all_objects;
Table created.

tkyte@TKYTE816> set timing on

tkyte@TKYTE816> update t set object_name = lower(object_name);

21946 rows updated.

Elapsed: 00:00:01.12
```

Многие, однако, по разным причинам предпочитают делать это так:

```
tkyte@TKYTE816> begin
  2      for x in  (select rowid rid, object_name, rownum r
  3                        from t)
  4      loop                 •
  5            update t
  6                set object name = lower(x.object_name)
  7              where rowid = x.rid;
  8            if  (mod(x.r,100)  = 0)  then
  9                commit;
 10           end if;
 11      end loop;
 12      commit;
 13  end;
 14  /
PL/SQL procedure successfully completed.

Elapsed: 00:00:05.99
```

На этом простом примере показано, что частое фиксирование изменений в цикле в пять раз **медленнее.** Если что-то можно сделать **одним** SQL-оператором, делайте это именно так — наверняка получится быстрее.

190 Глава 4

Теперь давайте вернемся ко второй причине, связанной с экономным использованием разработчиком "ограниченного ресурса" (сегментов отката). Это проблема конфигурирования — необходимо предусмотреть достаточно места в сегментах отката для поддержки транзакций требуемого размера. Фиксация в цикле — мало того, что обычно медленнее, это одна из наиболее частых причин возникновения печально известной ошибки **ORA-01555.** Давайте рассмотрим эту проблему более детально.

Как вы уже знаете из глав 1 и 3, в модели многовариантного доступа Oracle данные в сегментах отката используются для восстановления блоков в том виде, который они имели в начале выполнения оператора или транзакции (в зависимости от уровня изолированности). Если необходимые данные отмены недоступны, выдается сообщение об ошибке **ORA-01555 snapshot too old,** и запрос не выполняется. Так что, если вы изменяете считываемую вами таблицу (как в представленном выше анонимном блоке), то генерируете данные отмены, необходимые для выполнения запроса. Производимые изменения генерируют данные отмены, которые могут потребоваться при выполнении запроса для согласованного представления изменяемых данных. Если изменения фиксируются, системе разрешается повторно использовать только что занятое место в сегменте отката. Используя его, она стирает данные отката, которые в дальнейшем могут понадобиться для выполнения запроса, т.е. у вас возникает большая проблема. Оператор SELECT не выполнится, и изменение остановится на полпути. В результате получается частично завершенная транзакция, и никакого приемлемого способа выполнить ее повторно, как правило, не существует (об этом — чуть позже). Давайте рассмотрим это на маленьком примере. В небольшой тестовой базе данных я создал таблицу:

```
tkyte@TKYTE816> create table t as select * from all_objects;
Table created.
tkyte@TKYTE816> create index t_idx on t(object_name);
Index created.
```

Я затем отключил все сегменты отката и создал один маленький сегмент:

```
tkyte@TKYTE816> create rollback segment rbs_small storage (initial 64k
  2  next 64k minextents 2 maxextents 4) tablespace tools;

Rollback segment created.
```

Теперь, когда доступен только маленький сегмент отката, я ввел следующий блок кода для выполнения необходимых изменений:

```
tkyte@TKYTE816> begin
  2      for x in (select rowid rid, object_name, rownum r
  3                  from t
  4                 where object_name > chr(0))
  5      loop
  6          update t
  7             set object_name = lower(x.object_name)
  8           where rowid = x.rid;
  9          if (mod(x.r,100) = 0) then
 10             commit;
 11         end if;
```

```
 12      end loop;
 13      commit;
 14  end;
 15  /
begin
*
ERROR at line 1:
ORA-01555: snapshot too old: rollback segment number 10 with name
"RBS_SMALL" too small
ORA-06512: at line 2
```

Я получил сообщение об ошибке. Обращаю ваше внимание, что я добавил индекс и конструкцию **WHERE,** желая добиться случайности чтения строк таблицы. Конструкция **WHERE** приведет к использованию индекса (для этого я использовал оптимизатор, основанный на правилах). При доступе к таблице через индекс, обычно считывается блок, содержащий одну строку, а следующая необходимая строка оказывается уже в другом блоке. Мы обработаем все строки в блоке 1, просто не сразу. Пусть в блоке 1 находятся строки A, M, N, Q и Z. Так что придется обращаться к блоку пять раз через достаточно продолжительные интервалы времени. Поскольку изменения фиксируются часто и разрешено повторное использование пространства в сегменте отката, то в конце концов, повторно обратившись к блоку, который нельзя воспроизвести, мы получим сообщение об ошибке.

Этот придуманный пример наглядно демонстрирует, как все происходит. Оператор UPDATE генерировал данные отката. У нас было четыре экстента размером 64 Кбайта в сегменте отката, т.е. всего 256 Кбайт. Мы многократно использовали сегмент оката, поскольку сегменты отката используются циклически. При каждой фиксации серверу Oracle разрешалось перезаписывать сгенерированные нами данные отката. В определенный момент понадобился нами же сгенерированный фрагмент данных, которого больше нет, и было получено сообщение об ошибке **ORA-01555.**

Если не фиксировать изменения, то будет получено следующее сообщение об ошибке:

```
begin
*
ERROR at line 1:
ORA-01562: failed to extend rollback segment number 10
ORA-01628: max # extents (4) reached for rollback segment RBS_SMALL
ORA-06512: at line 6
```

Но между этими двумя ошибками есть, однако, важные отличия.

- Ошибка **ORA-01555 оставляет изменения в абсолютно неизвестном состоянии.** Часть изменений сделана, часть — нет.
- Мы **ничего** не можем **сделать, чтобы избежать ошибки ORA-01555,** если зафиксировали изменения в цикле **FOR** по курсору.
- **Всегда можно избежать ошибки ORA-01562,** выделив системе соответствующие ресурсы. Второй ошибки можно избежать, создав сегменты отката нужного размера, а первой — нет.

Подведу итоги: нельзя "сэкономить" место в сегментах отката, фиксируя изменения чаще, — эта информация в сегментах отката необходима. (Работая в однопользовательской системе, я в течение одного сеанса получил сообщение об ошибке **ORA-01555.)** Разработчики и администраторы баз данных должны совместно определить необходимый размер сегментов отката для обычно выполняемых заданий. Путем анализа системы необходимо выяснить максимальный размер транзакций и создать для них соответствующие сегменты отката. В рассмотренном выше примере это может быть однократное изменение. В данном случае можно отдельно создать в системе **большой** сегмент отката исключительно для выполнения этого изменения. Затем с помощью оператора SET **TRANSACTION** заставить транзакцию использовать этот сегмент отката. По ее завершении сегмент отката можно удалить. Если подобная транзакция выполняется неоднократно, необходимо включить сегмент отката необходимого размера в систему на постоянной основе. Многие считают такие компоненты, как временные пространства, сегменты отката и журналы повторного выполнения, излишним расходованием ресурсов и выделяют для них минимум места на диске. Это заставляет вспомнить проблему, с которой столкнулась компьютерная индустрия 1 января 2000 года — и все из-за стремления сэкономить 2 байта в поле даты. Эти компоненты базы данных — ключевые в системе, и они должны иметь соответствующий размер (не большой и не маленький, а такой, как нужно).

Конечно, самая большая проблема подхода с "фиксацией до завершения транзакции" состоит в том, что в случае прекращения изменений, база данных остается в неопределенном состоянии. Если заранее не подготовиться к этому, очень сложно продолжить выполнение неудачно завершившейся транзакции с момента останова. Например, если к столбцу применяется не функция **LOWER(),** а другая, например

```
last_ddl_time = last_ddl_time + 1;
```

Если цикл изменения прерван по ходу, как нам его перезапустить? Нельзя просто выполнить его повторно, поскольку к некоторым датам будет добавлено значение 2, а к некоторым — 1. При повторном сбое окажется, что к некоторым добавлено значение 3, к другим — 2, к остальным — I и так далее. Необходим более сложный подход, позволяющий "фрагментировать" данные. Например, можно обрабатывать все значения **object_names,** начинающиеся с A, затем — начинающиеся с B, и так далее:

```
tkyte@TKYTE816> create table to_do
  2  as
  3  select distinct substr(object_name, 1, 1) first_char
  4     from T
  5  /
Table created.
tkyte@TKYTE816> begin
  2             for x in (select * from to_do)
  3             loop
  4                  update t set last_ddl_time = last_ddl_time+1
  5                   where object_name like x.first char || '%';
  6
  7         dbms_output.put_line(sql%rowcount         || ' rows updated');
  8                 delete from to do where first char = x. first char;
```

```
  9
 10                  commit;
 11              end loop;
 12   end;
 13   /
11654 rows updated
21759 rows updated
309 rows updated
6 rows updated
270 rows updated
830 rows updated
412 rows updated
7 rows updated
378 rows updated
95 rows updated
203 rows updated
2482 rows updated
13 rows updated
318 rows updated
83 rows updated
14 rows updated
1408 rows updated
86 rows updated
2 rows updated
35 rows updated
2409 rows updated
57 rows updated
306 rows updated
379 rows updated
1 rows updated
1 rows updated

PL/SQL procedure successfully completed.
```

Теперь в случае сбоя мы сможем перезапустить процесс, поскольку не будут обрабатываться имена объектов, уже успешно обработанных. Проблема этого подхода, однако, в том, что данные неравномерно распределены по обрабатываемым фрагментам. Второе изменение затронуло больше строк, чем все остальные вместе взятые. Кроме того, другие сеансы, обращающиеся к таблице и изменяющие данные, тоже могут изменить столбец object_name. Предположим, другой сеанс изменил имя объекта с Z на A после обработки фрагмента, содержащего объекты с именами на A — соответствующая запись будет пропущена. Более того, это — крайне неэффективный процесс по сравнению с использованием оператора update t set last_ddl_time = last_ddl_time+1. Для чтения каждой строки таблицы мы обычно используем индекс или многократно просматриваем всю таблицу, однако и то, и другое нежелательно. У этого подхода слишком много недостатков.

Лучшим является подход, который я постоянно проповедую: делать проще. Если это можно сделать в SQL, делайте в SQL. То, что нельзя сделать одним SQL-оператором, делайте в PL/SQL. И используйте как можно меньше кода. Выделяйте достаточно ре-

сурсов. Всегда думайте, что произойдет в случае ошибки. Часто разработчики пишут циклы для изменения данных, замечательно работающие в тестовой среде, но завершающиеся ошибкой после применения к реальным данным. И проблема здесь в том, что не известно, где остановилась обработка. Намного проще определить правильный размер сегмента отката, чем написать перезапускаемую транзакцию. Если должны изменяться большие таблицы, фрагментируйте их (подробнее об этом см. в главе 14), что позволит изменять каждый фрагмент отдельно. Для изменения можно даже использовать параллельные операторы ЯМД.

И наконец, еще об одной плохой привычке при организации транзакций. Она приобретается при использовании популярных функциональных интерфейсов ODBC и JDBC. Эти функциональные интерфейсы по умолчанию выполняют "автоматическую фиксацию". Рассмотрим следующие операторы, переводящие 1000 $ с текущего счета на накопительный:

```
update accounts set balance = balance - 1000 where account_id = 123;
update accounts set balance = balance + 1000 where account_id = 456;
```

Если при эти операторы приложение отправляет через интерфейс JDBC, JDBC вставит оператор фиксации после **каждого** изменения. Подумайте о последствиях этого, если сбой произойдет после первого изменения, но перед вторым. Потеряны 1000 $!

Я понимаю, почему интерфейс ODBC делает именно так. Интерфейс ODBC создавали разработчики SQL Server, а принятая в этой СУБД модель одновременного доступа (пишущие блокируют читающих, читающие — пишущих, и блокировки являются ограниченным ресурсом) требует использования очень коротких транзакций. Я не понимаю, как подобный подход был перенесен в JDBC — интерфейс, предназначавшийся для поддержки приложений "масштаба предприятия". Практика показала, что сразу же после подключения по интерфейсу JDBC необходимо выполнять следующую строку кода:

```
connection conn81 = DriverManager.getConnection
                  ("jdbc:oracle:oci8:@ora8idev","scott","tiger");

conn81.setAutoCommlt (false);
```

Это возвращает разработчику контроль над транзакциями. Затем можно вполне безопасно выполнять транзакцию по переводу денег и фиксировать ее после успешного выполнения обоих операторов. Незнание особенностей используемого функционального интерфейса в данном случае просто опасно. Я знал разработчиков, не подозревавших об этом "свойстве" автоматической фиксации и столкнувшихся с большими проблемами, когда в созданных ими приложениях произошла подобная ошибка.

Распределенные транзакции

Одной из действительно замечательных возможностей сервера Oracle является его способность прозрачно для пользователей выполнять распределенные транзакции. Я могу изменять данные в нескольких базах в пределах одной транзакции. При фиксации эти изменения либо фиксируются во всех экземплярах, либо ни в одном (они все откатываются). Для этого не нужно писать дополнительный код — достаточно просто выполнить оператор фиксации.

Ключевым для распределенных транзакций в Oracle является понятие связи базы данных (database link). *Связь базы данных* — это объект базы данных, описывающий, как подключиться к другому экземпляру с текущего. Однако этот раздел посвящен не синтаксису оператора создания связи (он подробно описан в документации). После создания связей базы данных доступ к удаленным объектам выполняется очень просто:

```
select * from T@another_database;
```

Этот оператор выберет данные из таблицы T в экземпляре, определяемом связью базы данных **ANOTHER_DATABASE**. Обычно факт удаленности таблицы T "скрывают", создавая для нее представление или синоним. Например, можно выполнить:

```
create synonym T for T@another_database;
```

после чего обращаться к T, как к локальной таблице. Теперь, настроив связи базы данных и проверив их с помощью чтения ряда таблиц, мы сможем также изменять их (при наличии соответствующих привилегий, конечно). Выполнение распределенной транзакции теперь не отличается от выполнения локальной. Достаточно сделать следующее:

```
update local_table set x = 5;
update remote_table@another_database set y = 10;
commit;
```

И все. Сервер Oracle либо зафиксирует изменения в обеих базах данных, либо их не окажется ни в одной. Для этого используется протокол *двухэтапной фиксации* (two-phase commit — 2PC). 2PC — это распределенный протокол, позволяющий фиксировать неделимое изменение, затрагивающее несколько баз данных. Он пытается максимально ограничить возможность сбоя распределенной транзакции перед фиксацией. При использовании протокола 2PC одна из задействованных баз данных (обычно та, к которой первоначально подключен клиент) становится *координатором распределенной транзакции*. Сервер-координатор опрашивает соответствующие серверы, готовы ли они фиксировать изменения. Обращаясь к ним, он просит подготовиться к фиксации. Серверы сообщают о своем "состоянии готовности" в виде ДА или НЕТ. Если один из серверов сказал НЕТ, транзакция откатывается. Если все серверы сказали ДА, сервер-координатор рассылает сообщение о необходимости зафиксировать изменения на каждом сервере.

Этот протокол ограничивает время, когда может произойти серьезная ошибка. Перед "голосованием" по протоколу двухэтапной фиксации любая ошибка на задействованном сервере приведет к откату транзакции на всех серверах. Таким образом, не будет никаких сомнений как в отношении исхода транзакции, так и в том, что касается состояния распределенной транзакции. Сомнения в случае сбоя возможны только очень непродолжительное время — при сборе ответов. Предположим, например, что в транзакции участвует три сервера; сервер 1 — координатор. Сервер 1 попросил сервер 2 подготовиться к фиксации, и сервер 2 это сделал. Затем сервер 1 попросил сервер 3 подготовиться к фиксации, и тот подготовился. Сейчас только сервер 1 знает результат транзакции, и он обязан сообщить его двум другим серверам. Если в этот момент произойдет ошибка — сбой сети, сбой питания на сервере 1, любая ошибка, — серверы 2 и 3 останутся в "подвешенном" состоянии. У них возникнет так называемая сомнительная распределенная транзакция. Протокол двухэтапной фиксации пытается свести к ми-

нимуму время, когда такая ошибка может произойти, но оно, тем не менее, не равно нулю. Серверы 2 и 3 должны держать транзакцию открытой в ожидании уведомления о результате от сервера 1. Если вспомнить архитектуру, описанную в главе 2, решать эту проблему должен процесс **RECO.** Здесь также вступают в игру операторы **COMMIT** и **ROLLBACK** с опцией **FORCE.** Если причиной проблемы был сбой сети между серверами 1, 2 и 3, то администраторы базы данных серверов 2 и 3 могут просто позвонить администратору базы данных сервера 1 и, в зависимости от результатов, выполнить соответствующий оператор фиксации или отката вручную.

На действия, которые можно выполнять в распределенной транзакции, налагаются определенные ограничения. Этих ограничений, однако, немного, и все они обоснованы (мне, во всяком случае, они кажутся разумными). Вот основные ограничения.

- Нельзя выполнять оператор **COMMIT** в удаленной базе данных. Фиксировать транзакцию можно только на сервере-инициаторе.
- В удаленных базах данных нельзя выполнять операторы ЯОД. Это прямое следствие предыдущего ограничения на фиксацию операторов ЯМД. Фиксировать транзакцию можно только с сервера — инициатора транзакции, поэтому нельзя выполнять операторы ЯОД в связанных базах данных.
- В удаленных базах данных нельзя выполнять оператор **SAVEPOINT.** Короче, в удаленных базах данных нельзя выполнять операторы управления транзакцией.

Невозможность управлять транзакцией в удаленной базе данных вполне понятна, поскольку список серверов, задействованных в транзакции, есть только на сервере-инициаторе. В рассмотренной выше конфигурации с тремя серверами, если сервер 2 попытается зафиксировать транзакцию, он не сможет узнать, что в ней задействован сервер 3. В Oracle только сервер 1 может выполнять оператор фиксации. Но после этого сервер 1 может передать управление распределенными транзакциями другому серверу.

Повлиять на то, какой именно сервер будет фиксировать транзакцию, можно путем установки приоритета точки фиксации сервера (с помощью параметра в файле **init.ora).** *Приоритет точки фиксации* (commit point strength) задает для сервера относительный уровень важности в распределенной транзакции: чем важнее сервер (чем более доступными должны быть его данные), тем больше вероятность, что он будет координировать распределенную транзакцию. Это может пригодиться в том случае, если необходимо организовать распределенную транзакцию между тестовым и производственным сервером. Поскольку координатор транзакции **никогда** "не сомневается" в результатах транзакции, предпочтительнее, чтобы координировал такую транзакцию производственный сервер. Ничего страшного, если ряд открытых транзакций и заблокированных ресурсов останется на тестовой машине. На производственном сервере этого допускать нельзя.

Невозможность выполнять операторы ЯОД в удаленной базе данных, в общем, не так трагична. Во-первых, операторы ЯОД встречаются сравнительно редко. Они выполняются один раз после установки или после обновления программного обеспечения. На производственных системах в ходе работы операторы ЯОД не выполняются (по крайней мере **не должны** выполняться). Во-вторых, есть способ выполнить ЯОД в удаленной базе данных; для этого с помощью пакета **DBMS_JOB** (он подробно описан в при-

ложении А) формируется очередь заданий. Вместо того чтобы пытаться выполнить оператор ЯОД в удаленной базе данных, можно использовать связь для постановки в очередь задания, которое должно быть выполнено при фиксации транзакции. Такое задание выполняется на удаленной машине, но, не являясь распределенной транзакцией, вполне может включать операторы ЯОД. Именно так службы репликации Oracle выполняют распределенные операторы ЯОД при репликации схемы.

Журналы повторного выполнения и сегменты отката

Завершить эту главу о транзакциях я хочу описанием того, как генерируются данные повторного выполнения и отката (отмены), и как они используются в транзакциях, при восстановлении и т.д. Этот вопрос мне часто задают. Понимание журнала повторного выполнения и сегментов отката, а также того, что происходит в процессе работы, поможет разобраться в функционировании базы данных в целом. Глубокое понимание того, что происходит при изменении данных, — так же важно для разработчиков, как и для администраторов баз данных. Необходимо знать последствия своих действий. Ниже представлен псевдокод, описывающий действие этих механизмов в Oracle. На самом деле происходящее — несколько сложнее, но пример поможет разобраться в сути происходящего.

Посмотрим, что происходит при выполнении транзакции следующего вида:

```
insert into t (x,y) values (1,1);
update t set x = x+1 where x = 1;
delete from t where x = 2;
```

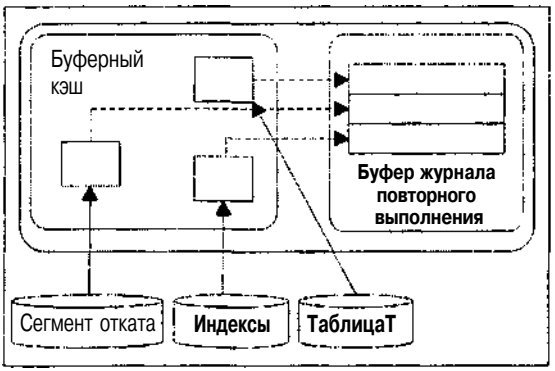
Разберем, что происходит при выполнении транзакции, по нескольким направлениям:

- что происходит, если возникает сбой после изменения?
- что происходит, если все операторы выполняются успешно и результаты фиксируются?
- что происходит при откате?

Первый оператор INSERT INTO T будет генерировать как данные повторного выполнения (redo), так и данные отмены (undo). Данных отмены будет достаточно, чтобы "избавиться" от последствий вставки. Данных повторного выполнения будет достаточно для того, чтобы повторить вставку. Данные отмены могут состоять из нескольких частей. Например, по столбцам X и Y могут быть индексы, и изменения в них также придется отменять при откате. Данные отмены хранятся в сегменте отката. Сегмент отката хранится в табличном пространстве и (это крайне важно) защищен журналом повторного выполнения, как и любой другой сегмент. Другими словами, данные отката обрабатываются так же, как и данные таблицы или индекса, — изменения в сегментах отката генерируют определенные данные повторного выполнения, которые записываются в журнал. (Зачем это делается, станет ясно после описания происходящего при сбое сие-

темы). Данные отмены добавляются в сегмент отката и кэшируются в буферном кэше, как и любые другие данные. Как и генерируемые данные отката, данные повторного выполнения могу состоять из нескольких частей.

Итак, после выполнения вставки мы имеем:



В кэше имеются измененные блоки сегмента отката, блоки индекса и блоки данных таблицы. Каждый из измененных блоков защищен записями в буфере журнала повторного выполнения. Вся эта информация пока кэширована в памяти.

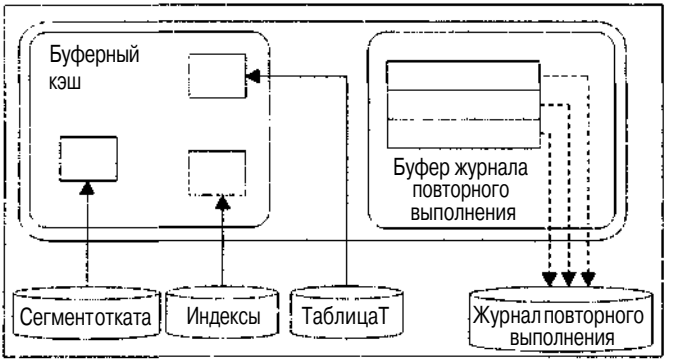
Гипотетический сценарий: **сбой системы происходит сейчас.** Все в порядке. Содержимое области SGA пропало, но хранившееся там нам не нужно. При перезапуске все пойдет так, будто транзакция никогда не выполнялась. Ни один из блоков с изменениями не сброшен на диск; не сброшены и данные повторного выполнения.

Гипотетический сценарий: **прямо сейчас заполняется буферный кэш.** Процесс **DBWR** должен освободить место и сбросить на диск только что измененные блоки. В этом случае процесс **DBWR** сначала попросит процесс **LGWR** сбросить на диск блоки журнала повторного выполнения, защищающие блоки базы данных. Прежде чем процесс **DBWR** сможет записать на диск измененные блоки, процесс **LGWR** должен сбросить на диск данные повторного выполнения, связанные с этими блоками. В этом есть смысл, так как, сбросив на диск измененные блоки таблицы Т и не сбросив при этом данные повторного выполнения для соответствующих блоков отмены, в случае сбоя системы мы получим измененный блок таблицы Т, для которого нет соответствующих данных отмены. Нужно сбросить на диск буферы журнала повторного выполнения, прежде чем записывать туда эти блоки, чтобы при необходимости отката можно было повторно выполнить все изменения для перевода области SGA в текущее состояние.

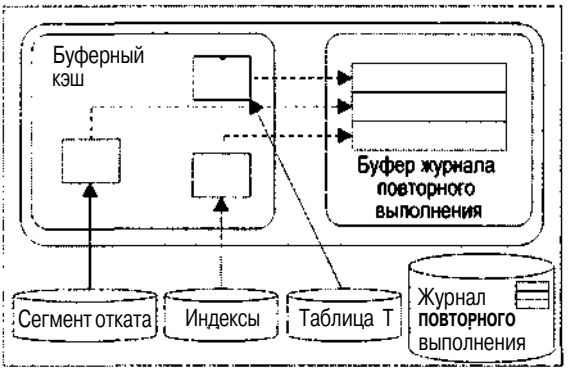
Второй сценарий поможет продемонстрировать, что привело к созданию подобной системы. Набор условий "если мы сбросили блоки таблицы Т, но не сбросили данные повторного выполнения для блоков отмены, **и** произошел сбой системы" становится сложным. При добавлении пользователей, дополнительных объектов и одновременной обработки все станет еще сложнее.

Итак, имеется ситуация, представленная на предыдущем рисунке. Мы сгенерировали ряд измененных блоков таблицы и индекса. В результате был создан ряд новых блоков в сегменте отката, и все три типа блоков сгенерировали определенный объем защи-

щающих их данных повторного выполнения. Если вспомнить буфер журнала повторного выполнения, обсуждавшийся ранее, — он сбрасывается каждые три секунды, при заполнении на треть и при обработке контрольной точки. Весьма вероятно, что в определенный момент буфер журнала повторного выполнения будет сброшен на диск, и некоторые из сделанных нами изменений тоже окажутся на диске. В этом случае схема приобретает следующий вид:



Теперь на диске могут оказаться и блоки, но мы этот случай не рассматриваем. Затем мы изменяем данные. Происходит примерно то же самое. В этот раз объем данных отмены больше (теперь в результате изменения нам надо сохранить ряд "предварительных" образов). Имеем следующую картину:



В буферном кэше добавились новые блоки сегмента отката. Чтобы при необходимости отменить изменение, можно воспользоваться имеющимися в кэше измененными блоками таблицы и индекса. Сгенерированы также дополнительные записи в буфер журнала повторного выполнения. Часть сгенерированных данных повторного выполнения находится на диске, часть — в кэше.

Гипотетический сценарий: **сбой системы происходит в этот момент.** При запуске сервер Oracle будет считывать журналы повторного выполнения и обнаружит ряд записей повторного выполнения транзакции. С учетом состояния системы (записи повторного выполнения для оператора вставки находятся в файлах журнала на диске, а записи повторного выполнения для оператора изменения — все еще в буфере) сервер Oracle "на-

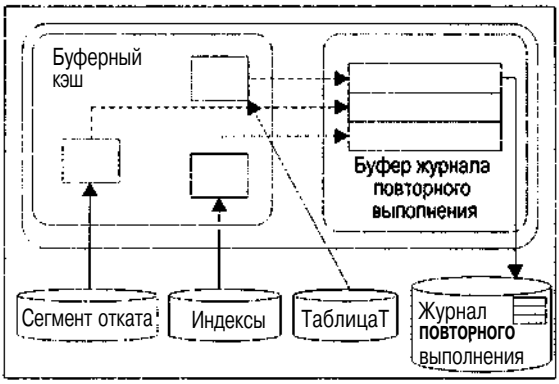
катит" (roll forward) вставку. В результате получится схема, аналогичная первой: в памяти имеется ряд блоков отмены из сегмента отката (для отмены вставки), измененные (после вставки) блоки таблицы и измененные (после вставки) блоки индекса. Теперь сервер Oracle обнаружит, что транзакция не зафиксирована, и откатит ее, поскольку выполняется восстановление после сбоя, а наш сеанс, конечно же, не подключен. Сервер прочитает данные отмены, помещенные при накате в буферный кэш, и применит к блокам данных и индексов, приводя их к состоянию, которое было до вставки. Все вернулось на свои места. Блоки на диске могут отражать (или не отражать) результаты выполнения оператора **INSERT** (это зависит от того, были ли они сброшены на диск перед сбоем). Если были, значит, вставка отменена, и при сбросе блоков из буферного кэша на диск эта отмена будет отражена и в файлах данных. Если же результатов вставки в этих блоках нет, — пусть так и будет, их все равно потом перезапишут.

Гипотетический сценарий: **приложение откатывает транзакцию. В** этот момент сервер Oracle находит данные отмены для транзакции либо в блоках сегмента отката в кэше (скорее всего), либо на диске, если они уже были сброшены (это более вероятно в случае очень больших транзакций). Он применяет данные отмены к блокам данных и индекса в буферном кэше; если их в кэше уже нет, они считываются с диска в кэш, после чего к ним применяются данные отмены. Эти блоки в дальнейшем будут сброшены в файлы данных в исходном виде.

Первый сценарий затрагивает некоторые детали восстановления в случае сбоя. Сервер выполняет его за два шага. Сначала он накатывает изменения, приводя систему к состоянию на момент сбоя, после этого откатывается все, что не было зафиксировано. Это действие повторно синхронизирует файлы данных. Сервер повторно выполняет все действия, а затем отменяет все, что не было завершено.

Со вторым сценарием мы сталкиваемся намного чаще. Следует помнить, что в ходе отката журналы повторного выполнения никогда не используются. Журналы повторного выполнения читаются только при восстановлении и архивировании. Это ключевая концепция настройки: журналы повторного выполнения — только для записи. В ходе обычной обработки сервер Oracle их не читает. Если имеется достаточное количество устройств, т.е. при считывании процессом **ARCH** файла процесс **LGWR** выполняет запись на другое устройство, конфликты доступа к журналам повторного выполнения не возникают. Во многих других СУБД файлы журналов используются, как "журналы транзакций". В них нет разделения данных повторного выполнения и отмены — и те, и другие хранятся в одном файле. Для таких систем откат может оказаться катастрофой: в процессе отката должны считываться журналы, в которые выполняет запись процесс, сбрасывающий буферы журнала. В результате конфликты возникают в той части системы, где они наименее желательны. Разработчики Oracle постарались сделать так, чтобы журналы записывались последовательно и никто не читал бы их в процессе записи.

Теперь переходим к оператору **DELETE.** И в этом случае генерируются данные отмены, блоки изменяются, а записи повторного выполнения отправляются в буфер журнала повторного выполнения. Это мало отличается от обработки оператора **UPDATE,** рассмотренной ранее, поэтому перейдем к оператору **COMMIT.** При обработке этого оператора сервер Oracle сбросит буфер журнала повторного выполнения на диск, и схема происходящего будет выглядеть примерно так:



Измененные блоки находятся в буферном кэше; возможно, некоторые из них были сброшены на диск. Все данные повторного выполнения, необходимые для восстановления транзакции, находятся в безопасности на диске, и теперь изменения стали постоянными. Если придется считывать данные непосредственно из файлов данных, то блоки будут получены в том состоянии, в котором они находились **перед** началом транзакции, поскольку процесс **DBWR** их скорее всего еще не записал. Это нормально: в случае сбоя записи в журнале повторного выполнения можно будет использовать для восстановления блоков. Данные отмены будут сохраняться еще некоторое время — до повторного использования сегмента отката и перезаписи соответствующих блоков. Сервер Oracle с помощью этих данных отмены обеспечит согласованное чтение измененных объектов любыми сеансами, которым это надо.

Резюме

В этой главе мы рассмотрели различные аспекты управления транзакциями в Oracle. Поддержка транзакций — одна из основных особенностей, отличающих базу данных от файловой системы. Понимание их механизма и способов использования необходимо для создания корректно работающих приложений для любой СУБД. Понимание того, что в Oracle все операторы (включая их побочные эффекты) — неделимы и что эта неделимость распространяется на хранимые процедуры, имеет принципиальное значение. Мы показали, как добавление обработчика исключительных ситуаций **WHEN OTHERS** в блок PL/SQL может радикально повлиять на изменения в базе данных. Для разработчиков приложений глубокое понимание работы транзакций принципиально важно.

Мы рассмотрели весьма сложное взаимодействие требований целостности (уникальных ключей, требований проверки и т.д.) и транзакций в Oracle. Было описано, что сервер Oracle обычно обрабатывает требования целостности сразу после выполнения оператора, но при желании можно отложить проверку требований до конца транзакции. Эта возможность является ключевой при выполнении сложных многотабличных изменений, когда изменяемые таблицы — взаимозависимы. В качестве примера рассматривалось каскадное изменение.

Затем были описаны вредные привычки, вырабатываемы в процессе работы с базами данных, "поддерживающими", но не "навязывающими" использование транзакций.

Изложено основное правило организации транзакций: они могут быть короткими или длинными — это определяется необходимостью. Размер транзакции определяется требованием целостности данных — вот основная мысль, которую мы пытались донести до вас в этой главе. Только установленные в системе правила обработки данных должны определять размер транзакций, но никак не размер сегментов отката и не количество блокировок.

Рассмотрены распределенные транзакции и их отличие от транзакций, затрагивающих только одну базу данных. Описаны ограничения, налагаемые на распределенные транзакции, и объяснены причины этого. Прежде чем создавать распределенную систему, необходимо понимать эти ограничения. То, что работает на одном сервере, может не работать в распределенной базе данных.

Глава завершается описанием использования данных повторного выполнения и отмены, архитектурных особенностей, позволяющих обеспечить обсуждавшиеся в начале главы свойства транзакций (ACID). Изменение данных в транзакции было описано с точки зрения генерации данных для повторного выполнения и отмены (данных отката). В следующей главе это описывается очень подробно.

5

Повторное выполнение и откат

В главе 4 мы рассмотрели основы повторного выполнения и отката (отмены) транзакций. Там было описано, что такое данные повторного выполнения. Это данные, которые сервер Oracle записывает в активные файлы журнала для повторного выполнения транзакций в случае сбоя. Они позволяют серверу Oracle восстанавливать выполненные транзакции. Мы также рассматривали данные отмены, или отката, которые сервер Oracle записывает в сегменты отката для отмены, или отката, транзакции. Кроме того, мы затронули несколько проблем, например причины возникновения ошибки **ORA-01555: snapshot too old** или прекращения обработки контрольной точки **(checkpoint not complete, cannot allocate new log). В** этой главе я хочу более глубоко рассмотреть суть процессов повторного выполнения и отката, а также описать, что должны знать о них разработчики.

Повторное выполнение и откат — это темы, одинаково важные для администратора базы данных и для разработчиков. И тем, и другим необходимо глубокое понимание их назначения, алгоритмов работы, а также знание того, как избегать связанных с этими процессами проблем. Эта информация будет представлена в данной главе. Мы не будем касаться средств, использование и настройка которых является исключительной прерогативой администратор базы данных. Например, мы не будем рассматривать, как найти оптимальное значение параметров инициализации **RECOVERY_PARALLELISM** или **FAST_START_IO_TARGET.** Мы сконцентрируемся на проблемах, о которых должен знать разработчик, и на том, как они могут влиять на приложение.

Повторное выполнение

Файлы журнала повторного выполнения чрезвычайно важны для базы данных Oracle. Это журналы транзакций базы данных. Они используются только для восстановления; единственное их назначение — предоставить необходимую информацию в случае сбоя экземпляра или носителя. Если на машине, где работает сервер, пропадает питание, что приводит к сбою экземпляра, сервер Oracle будет использовать активные журналы повторного выполнения для восстановления системы в состояние на момент, непосредственно предшествующий отключению питания. Если происходит сбой диска, сервер Oracle будет использовать архивные журналы повторного выполнения для восстановления резервной копии этого диска на соответствующий момент времени. Кроме того, если случайно удалена таблица или важные данные и это изменение зафиксировано, можно восстановить потерянные данные с резервной копии, а затем с помощью активного и архивных файлов журнала повторного выполнения привести их в состояние, соответствующее моменту, непосредственно предшествующему этому случаю.

Сервер Oracle поддерживает два типа файлов журнала повторного выполнения: активные и архивные. В каждой базе данных Oracle есть по крайней мере два активных файла журнала повторного выполнения. Эти активные файлы журнала повторного выполнения используются циклически. Сервер Oracle будет выполнять запись в журнальный файл 1, а добравшись до конца этого файла, переключится на журнальный файл 2 и начнет записывать в него. Заполнив журнальный файл 2, он опять переключится на журнальный файл 1 (если файлов журнала повторного выполнения всего два; если их три, сервер, конечно же, переключится на третий файл). Архивные файлы журнала повторного выполнения — это просто копии старых, заполненных активных файлов журнала повторного выполнения. Когда система заполняет журнальные файлы, процесс ARCH копирует активный файл журнала повторного выполнения в другое место. Архивные файлы журнала повторного выполнения используются для восстановления носителя, когда сбой приводит к порче диска или в случае другого физического сбоя. Сервер Oracle может применять эти архивные файлы журнала повторного выполнения к резервным копиям файлов данных, чтобы привести их в соответствие с остальной базой данных. Эти журналы содержат хронологию транзакций в базе данных.

Поддержка журналов повторного выполнения, или журналов транзакций, — одна из основных особенностей баз данных. Эти журналы, пожалуй, — самая важная структура, обеспечивающая восстановление, хотя без остальных компонентов, таких как сегменты отката, средства восстановления распределенных транзакций и т.п. тоже ничего не получится. Именно эти основные компоненты отличают базу данных от обычной файловой системы. Активные журналы повторного выполнения позволяют эффективно восстанавливать данные после сбоя питания, которое может произойти в тот момент, когда сервер Oracle выполняет запись. Архивные журналы повторного выполнения позволяют восстановить данные в случае выхода из строя носителей, например в случае поломки жесткого диска. Без журналов повторного выполнения база данных обеспечивала бы не большую защиту, чем файловая система.

Важно понять, какое значение файлы журнала имеют для разработчиков. Мы рассмотрим, как различные способы написания кода влияют на использование журналов.

Разберемся, почему возникают некоторые ошибки в базе данных (в частности, ORA-01555: snapshot too old) и как их предотвратить. Мы уже рассматривали механизм повторного выполнения в главе 4, а теперь затронем ряд специфических проблем. Многие из этих проблем могут выявить разработчики, но исправлять их должен администратор базы данных, поскольку они влияют на весь сервер в целом. Мы начнем с анализа происходящего в ходе фиксации, а затем перейдем к часто задаваемым вопросам и проблемам, связанным с активными журналами повторного выполнения.

Что происходит при фиксации?

Разработчики должны очень четко понимать, что происходит при выполнении оператора COMMIT. Фиксация транзакции — очень быстрая операция, независимо от размера транзакции. Можно подумать, что чем больше транзакция (т.е. чем больше данных она затрагивает), тем дольше будет выполняться фиксация. Это неверно. Время выполнения фиксации обычно достаточно стабильно и мало зависит от размера транзакции. Дело в том, что при фиксации не так уж много делается, но эти действия — жизненно важны.

Понимая все это, вы сможете делать транзакции настолько большими, как они должны быть. Многие разработчики искусственно ограничивают размер своих транзакций, фиксируя строки группами, а не после выполнения логической единицы работы. Они делают это, исходя из ошибочного предположения, что экономят ресурсы системы; на самом же деле они их транжирят. Если фиксация одной строки требует X единиц времени, а фиксация тысячи строк — тех же X единиц времени, то, выполняя работу так, что 1000 строк фиксируется одним оператором COMMIT, можно сэкономить 999 единиц времени. Фиксируя транзакции только при необходимости (когда транзакция закончена), вы не только повышаете производительность, но и сокращаете конфликты доступа к общим ресурсам (журнальным файлам, внутренним защелкам и т.п.). Простой пример демонстрирует, что короткие транзакции требуют для выполнения больше времени:

```
tkyte@TKYTE816> create table t (x int) ;
tkyte@TKYTE816> set serveroutput on
tkyte@TKYTE816> declare
  2             l_start number default dbms_utility.get_time;
  3            begin
  4           for i in 1 . . 1000
  5            loop
  6                     insert into t values (1) ;
  7            end loop;
  8            commit;
  9            dbms_output.put_line
 10            (dbms_utility.get_time-l_start || ' hsecs');
 11  end;
 12  /
7 hsecs

PL/SQL procedure successfully completed.

tkyte@TKYTE816> declare
```

```
  2            l_start number default dbms_utility.get_time;
  3  begin
  4          for i in 1 .. 1000
  5            loop
  6                     insert into t values (1) ;
  7                      commit;
  8            end loop;
  9            dbms_output.put_line
 10           (dbms_utility.get_time-l_start || ' hsecs');
 11  end;
 12  /
21 hsecs

PL/SQL procedure successfully completed.
```

В данном случае потребовалось в три раза больше времени (в вашем случае результат может быть другим). При одновременном выполнении этого кода несколькими пользователями, слишком часто фиксирующими транзакции, скорость работы существенно уменьшается. Мной было показано, как не использование связываемых переменных и частое выполнение жесткого разбора заметно снижает параллелизм из-за конфликтов доступа к библиотечному кэшу и дополнительной нагрузки на процессор. Даже с учетом использования связываемых переменных слишком частое выполнение мягких разборов существенно увеличивает расходы ресурсов. Операции надо выполнять, только когда это действительно необходимо, а фиксация транзакции — такая же операция, как и разбор. Размер транзакций должен определяться требованиями приложения, а не ошибочными попытками сэкономить ресурсы базы данных.

Так почему же продолжительность фиксации почти не зависит от размера транзакции? До начала фиксации изменений в базе данных, мы уже сделали все самое сложное — изменили данные, так что 99,9 процента работы сделано. Например, уже были выполнены следующие операции:

- в области SGA сгенерированы записи сегмента отката;
- в области SGA сгенерированы измененные блоки данных;
- помещены в буфер журнала повторного выполнения в области SGA данные повторного выполнения для перечисленных выше изменений;
- в зависимости от размера трех предыдущих фрагментов данных и прошедшего времени, часть их может уже быть сброшена на диск;
- установлены все необходимые блокировки.

При выполнении фиксации осталось сделать следующее.

- Сгенерировать номер системного изменения (SCN — System Change Number) для транзакции.
- Процессу **LGWR** надо записать на диск все **оставшиеся** записи из буфера журнала повторного выполнения, а также записать в активные файлы журнала повторного выполнения номер SCN. Именно этот шаг и является фактической фиксацией. Если этот шаг выполнен, — транзакция зафиксирована. Если запись

транзакции удалена, значит, транзакция зафиксирована. Соответствующая запись в представлении **V$TRANSACTION** исчезнет.

- Все блокировки, удерживаемые транзакцией, снимаются, и все сеансы, ожидавшие в очередях снятия этих блокировок, могут продолжить работу.
- Многие измененные транзакцией блоки данных будут повторно обработаны и "очищены" в быстром режиме, если они еще находятся в буферном кэше.

Как видите, для обработки оператора **COMMIT** надо сделать очень немного. Самая продолжительная операция выполняется процессом **LGWR,** поскольку связана с физическим вводом/выводом. Время работы процесса **LGWR** при этом будет в рамках допустимого (ограничено), поскольку он периодически сбрасывает на диск содержимое буфера журнала повторного выполнения. Процесс **LGWR** не буферизует все изменения по ходу их выполнения. Он постепенно сбрасывает содержимое буфера журнала повторного выполнения в фоновом режиме по мере его заполнения. Так делается, чтобы при выполнении оператора COMMIT не пришлось очень долго ждать разового сброса всей сгенерированной информации повторного выполнения. Процесс **LGWR** выполняет этот сброс постоянно:

- каждые три секунды;
- при заполнении буфера журнала на треть или при записи в него 1 Мбайт информации;
- при фиксации транзакции.

Так что, даже если транзакция выполняется долго, большая часть сгенерированной информации повторного выполнения уже сброшена на диск еще до фиксации. С другой стороны, однако, при фиксации необходимо дождаться, пока **вся** оставшаяся в буфере сгенерированная информация повторного выполнения не будет сохранена на диске. Таким образом, обращение к процессу **LGWR** выполняется синхронно. Хотя процесс LGWR и может использовать асинхронный ввод/вывод для параллельной записи в файлы журналов повторного выполнения, транзакция будет ждать, пока процесс **LGWR** не закончит все записи и не получит подтверждение записи всех данных на диск.

Если вам еще не знаком упомянутый выше номер SCN — это простой механизм отслеживания времени, используемый сервером Oracle для упорядочения транзакций и возможности восстановления после сбоя. Он также используется для обеспечения согласованности по чтению и при обработке контрольной точки в базе данных. Номер SCN можно рассматривать как счетчик; при каждой фиксации транзакции значение SCN увеличивается на единицу.

Осталось прояснить одно: что значит "очистить" блоки базы данных. Я написал, что некоторые блоки, измененные транзакцией, будут повторно обработаны и очищены в процессе фиксации. Имеется в виду информация о блокировках, хранящаяся в заголовках блоков данных. Далее, в разделе "Очистка блоков", этот процесс будет описан подробно. Если коротко, происходит удаление данных транзакции из блока, чтобы следующий посетитель блока был избавлен от необходимости делать это. При этом не генерируются данные в журнал повторного выполнения, что существенно экономит ресурсы.

Чтобы продемонстрировать, что продолжительность фиксации не зависит от размера транзакции, я буду генерировать различные объемы данных повторного выполнения и замерять время работы операторов **INSERT** и **COMMIT.** Для этого необходимо получить привилегии доступа к представлениям V$ (подробнее об этих представлениях см. в главе 10). Получив эти привилегии, мы создадим достаточно большую тестовую таблицу. В данном случае для генерации строк данных я использую представление **ALL_OBJECTS** и вставлю несколько экземпляров соответствующих строк, чтобы в итоге получилось порядка 100000 строк для работы (для получения такого же количества строк вам, возможно, придется выполнить операторы **INSERT** другое количество раз):

```
tkyte@TKYTE816> connect sys/change_on_install
sys@TKYTE816> grant select on v_$mystat to tkyte;
Grant succeeded.
sys@TKYTE816> grant select on v_$statname to tkyte;
Grant succeeded.
sys@TKYTE816> connect tkyte/tkyte
tkyte@TKYTE816> drop table t;
Table dropped.
tkyte@TKYTE816> create table t
  2  as
  3  select * from all_objects
  4  /
Table created.
tkyte@TKYTE816> insert into t select * from t;
21979 rows created.
tkyte@TKYTE816> insert into t select * from t;
43958 rows created.
tkyte@TKYTE816> insert into t select * from t where rownum < 12000;
11999 rows created.
tkyte@TKYTE816> commit;
Commit complete.
tkyte@TKYTE816> create or replace procedure do_commit(p_rows in number)
  2  as
  3        l_start         number;
  4        l_after_redo    number;
  5        l_before_redo   number;
  6  begin
  7        select v$mystat.value into l_before_redo
  8          from v$mystat, v$statname
  9         where v$mystat.statistic# = v$statname.statistic#
 10           and v$statname.name = 'redo size';
 11
 12        l_start := dbms_utility.get_time;
 13       insert into t select * from t where rownum < p_rows;
```

```
 14        dbms_output.put_line
 15        (sql%rowcount || ' rows created');
 16        dbms_output.put_line
 17        ('Time to INSERT: ' ||
 18          to_char(round((dbms_utility.get_time-l_start)/100, 5),
 19                  '999.99') ||
 20         ' seconds');
 21
 22        l_start := dbms_utility.get_time,-
 23        commit;
 24        dbms_output.put_line
 25        ('Time to COMMIT: ' ||
 26          to_char(round((dbms_utility.get_time-l_start)/100, 5),
 27                   '999.99') ||
 28         ' seconds');
 29
 30        select v$mystat.value into l_after_redo
 31          from v$mystat, v$statname
 32         where v$mystat.statistic# = v$statname.statistic#
 33           and v$statname.name = 'redo size';
 34
 35        dbms_output.put_line
 36        ('Generated ' ||
 37      to_char(l_after_redo-l_before_redo,'999,999,999,999')      ||
 38         ' bytes of redo') ;
 39        dbms_output.new_line;
 40 end;
 41 /

Procedure created.
```

Теперь все готово для демонстрации последствий фиксации транзакций разного размера. Мы будем вызывать созданную ранее процедуру, требуя создать различное количество строк и проинформировать о результатах:

```
tkyte@TKYTE816> set serveroutput on format wrapped
tkyte@TKYTE816> begin
  2       for i in 1 . . 5
  3        loop
  4            do_commit(power(10,i)) ;
  5        end loop;
  6   end;
  7   /
9 rows created
Time to INSERT:       .06 seconds
Time to COMMIT:       .00 seconds
Generated              1,512 bytes of redo

99 rows created
Time to INSERT:       .06 seconds
Time to COMMIT:       .00 seconds
Generated             11,908 bytes of redo
```

```
999 rows created
Time to INSERT:       .05 seconds
Time to COMMIT:       .00 seconds
Generated            115,924 bytes of redo

9999 rows created
Time to INSERT:       .46 seconds
Time to COMMIT:       .00 seconds
Generated          1,103,524 bytes of redo

99999 rows created
Time to INSERT:     16.36 seconds
Time to COMMIT:       .00 seconds
Generated         11,220,656 bytes of redo


PL/SQL procedure successfully completed.

tkyte@TKYTE816> show parameter log_buffer

NAME                                 TYPE     VALUE

log buffer                           integer  512000
```

Как видите, при генерации разного объема данных повторного выполнения, от 1512 байт до 11220656 байт, время выполнения оператора **COMMIT** пренебрежимо мало — менее одной сотой секунды. Я выдавал время выполнения операторов **INSERT** специально, чтобы продемонстрировать, что алгоритм таймера "работает". Если делается что-то, занимающее определенное время, таймер это показывает, просто операторы **COMMIT** выполняются слишком быстро. По ходу обработки, по мере генерации данных в журнал повторного выполнения процесс **LGWR** постоянно сбрасывал данные из буфера журнала на диск в фоновом режиме. Так что, даже когда мы сгенерировали 11 Мбайт данных повторного выполнения, процесс **LGWR** постоянно сбрасывал ее на диск порциями примерно по 170 Кбайт (треть от 512000 байт). Когда дошло до выполнения оператора **COMMIT,** осталось сделать не так уж много — не намного больше, чем при вставке девяти строк данных. Можно ожидать подобных (но не точно таких же) результатов, независимо от объема сгенерированных данных повторного выполнения.

Что происходит при откате?

Если заменить оператор COMMIT оператором ROLLBACK результат будет совершенно другим. Время отката будет зависеть от объема измененных данных. Я изменил созданную в предыдущем разделе процедуру DO_COMMIT так, чтобы она выполняла откат (просто заменил оператор COMMIT в строке 23 оператором ROLLBACK) и получил совсем другое время выполнения. Например:

```
9 rows created
Time to INSERT:       .06 seconds
Time to ROLLBACK:       .02 seconds
Generated              1,648 bytes of redo
```

```
99 rows created
Tine to INSERT:       .04 seconds
Time to ROLLBACK:       .00 seconds
Generated             12,728 bytes of redo

999 rows created
Time to INSERT:       .04 seconds
Time to ROLLBACK:       .01 seconds
Generated            122,852 bytes of redo

9999 rows created
Time to INSERT:      . 94 seconds
Time to ROLLBACK:       .08 seconds
Generated          1,170,112 bytes of redo

99999 rows created
Time to INSERT:     8.08 seconds
Time to ROLLBACK:     4.81 seconds
Generated         11,842,168 bytes of redo

PL/SQL procedure successfully completed.
```

Это вполне можно было предсказать, поскольку при откате приходится физически отменять сделанные изменения. Как и при фиксации, необходимо выполнить ряд операций. Прежде чем добраться до оператора ROLLBACK, сервер уже многое сделал. Напомню, что было сделано следующее:

- в области SGA сгенерированы записи сегмента отката;
- в области SGA сгенерированы измененные блоки данных;
- помещены в буфер журнала повторного выполнения в области SGA данные повторного выполнения для перечисленных выше изменений;
- в зависимости от размера этих трех фрагментов информации и прошедшего времени, часть их может уже быть сброшена на диск;
- установлены все необходимые блокировки.

При откате:

- Отменяются все сделанные изменения. Это достигается путем считывания данных из сегмента отката и выполнения обратной операции. Если строка была вставлена, при откате она будет удалена. Если строка изменена, при откате будут восстановлены прежние значения столбцов. Если строка была удалена, при откате она будет снова вставлена.
- Снимаются все блокировки, удерживаемые сеансом, и все сеансы, ждавшие в очереди снятия этих блокировок, могут продолжить работу.

При фиксации же просто сбрасываются на диск все данные, оставшиеся в буфере журнала повторного выполнения. По сравнению с откатом, действий при этом выполняется очень мало. Идея в том, что откатывать транзакцию не надо, если в этом нет крайней необходимости. Это влечет большие расходы ресурсов системы, поскольку сна-

чала много времени уходит на выполнение изменений, а потом — на их отмену. Не делайте никаких изменений, если не собираетесь их фиксировать. Это кажется очевидным: естественно, никто ничего не делает, если не собирается результат фиксировать. Я, однако, неоднократно видел, как разработчики используют "реальную" таблицу в качестве временной, наполняя ее данными, а затем откатывая изменения. Ниже мы еще обсудим реальные временные таблицы и как избежать подобной проблемы.

Какой объем данных повторного выполнения генерируется?

Разработчикам может потребоваться определить объем генерируемых операциями данных повторного выполнения. Чем больше данных повторного выполнения генерируется, тем дольше будут выполняться операции и тем медленнее будет работать система в целом. Вы влияете не только на свой сеанс, но и на все сеансы. Управление данными повторного выполнения в базе данных осуществляется последовательно — рано или поздно все сеансы обращаются к процессу LGWR, чтобы он обработал их данные повторного выполнения и зафиксировал транзакцию. Чем больше ему надо сделать, тем медленнее будет работать система. Зная, сколько информации повторного выполнения генерируется, и проверяя несколько подходов к решению проблемы, можно найти наилучший вариант.

Как было показано выше, определить объем генерируемой информации повторного выполнения достаточно просто. Я использовал представление динамической производительности V$MYSTAT, содержащее статистику только моего сеанса, и соединил его с представлением V$STATNAME. Я выбрал из него значение статистического показателя redo size. Мне не пришлось угадывать имя этого показателя — его легко найти в представлении V$STATNAME:

```
ops$tkyte@DEV816> select * from v$statname
  2  where name like 'redo%';

STATISTIC#  NAME                                CLASS

        61  redo synch writes                       8
        62  redo synch time                         8
        98  redo entries                            2
        99  redo                 size               2
       100  redo buffer allocation retries          2
       101  redo wastage                            2
       102  redo writer latching time               2
       103  redo writes                             2
       104  redo blocks written                     2
       105  redo write time                         2
       106  redo log space requests                 2
       107  redo log space wait tine                2
       108  redo log switch interrupts              2
       109  redo ordering marks                     2

14 rows selected.
```

Теперь можно приступать к изучению способа определения объема генерируемых транзакцией данных повторного выполнения. Достаточно легко оценить, сколько данных повторного выполнения будет сгенерировано, если известно, сколько данных будет изменено. Ниже я создаю таблицу, длина строки которой — около 2010 байт, плюс-минус несколько байт. Поскольку данные типа **CHAR** всегда имеют максимальный размер, в строке 2000 байт занимает столбец типа **CHAR,** 7 байт — столбец типа DATE, и три байта необходимо для представления числа — всего получается около 2010 байт, плюс некоторое количество" байтов на представление строки:

```
tkyte@TKYTE816> create table t (x int, y char(2000), z date);

Table created.
```

Рассмотрим, сколько данных повторного выполнения будет сгенерировано при вставке, затем — при изменении и, наконец, — при удалении одной, десяти и множества таких строк. Разберемся также, есть ли существенное различие между одновременным изменением строк и по одной, основываясь на объеме генерируемых данных повторного выполнения. Мы уже знаем, что изменение строк по одной выполняется медленнее, чем с помощью одного оператора **UPDATE.**

Для измерения объема генерируемых данных повторного выполнения можно использовать средство **AUTOTRACE** утилиты SQL*Plus либо непосредственный запрос к представлениям **V$MYSTAT/V$STATNAME,** показывающий объем данных повторного выполнения сеанса:

```
tkyte@TKYTE816> create or replace view redo_size
  2  as
  3  select value
  4    from v$mystat, v$statname
  5   where v$mystat.statistic# = v$statname.statistic#
  6     and v$statname.name = 'redo size';

View created.
```

Для операторов, трассируемых при установке **AUTOTRACE** (операторов **INSERT, UPDATE** и **DELETE),** мы будем использовать выдаваемую информацию. Для блоков PL/SQL придется обращаться к представлениям V$MYSTAT/V$STATNAME, поскольку **AUTOTRACE** для них такую информацию генерировать не будет.

Для оценки объема генерируемых данных повторного выполнения будем использовать созданную ранее таблицу T со сравнительно постоянным размером строки 2010 байт, если все столбцы не могут иметь пустых значений. Будем выполнять различные операции и измерять объем сгенерированных при этом данных повторного выполнения. Вставим сначала одну строку, затем — десять строк с помощью одного оператора, затем — 200 строк также с помощью одного оператора и, наконец, 200 строк — по одной. Аналогичную проверку выполним для изменения и удаления строк.

Ниже представлен код для этого примера. Вместо обычного копирования и вставки кода непосредственно из утилиты **SQL*Plus,** рассмотрим использованные операторы, а затем — таблицу итоговых результатов:

```
set autotrace traceonly statistics
insert into t values (1, user, sysdate);

insert into t
select object_id, object_name, created
  from all_objects
 where rownum <= 10;

insert into t
select object_id, object_name, created
  from all_objects
 where rownum <= 200
/

declare
  l_redo_size number;
  l_cnt        number := 0;
begin
    select value into l_redo_size from redo_size;
    for x in (select * from all_objects where rownum <= 200)
    loop
        insert into t values
        (x.object_id, x.object_name, x.created);
        l_cnt := l_cnt+1;
    end loop;
    select value-l_redo_size into l_redo_size from redo_size;
    dbms_output.put_line('redo size = ' || l_redo_size ||
                         ' rows = ' || l_cnt);
end;
/
```

Этот фрагмент кода выполняет описанные операторы вставки — 1, 10, 200 строк за раз, **затем** — 200 отдельных операторов **INSERT.** Выполним изменения:

```
update t set y=lower(y) where rownum = 1;

update t set y=lower(y) where rownum <= 10;

update t set y=lower(y) where rownum <= 200;

declare
  l_redo_size number;
  l_cnt        number := 0;
begin
  select value into l_redo_size from redo_size;
  for x in (select rowid r from t where rownum <= 200)
  loop
      update t set y=lower(y) where rowid = x.r;
      l_cnt := l_cnt+1;
  end loop;
  select value-l_redo_size into l_redo_size from redo_size;
  dbms_output.put_line('redo size = ' || l_redo_size ||
                       ' rows = ' || l_cnt);
end;
/
```

и, наконец, удаления:

```
delete from t where rownum = 1;

delete from t where rownum <= 10;

delete from t where rownum <= 200;

declare
   l_redo_size number;
   l_cnt        number := 0;
begin
    select value into l_redo_size from redo_size;
    for x in (select rowid r from t) loop
        delete from t where rowid = x.r;
        l_cnt := l_cnt+1;
    end loop;
    select value-l_redo_size into l_redo_size from redo_size;
    dbms_output.put_line('redo size = ' || l_redo_size ||
                         ' rows = ' || l_cnt);
end;
```

Вот результаты эксперимента:

Операция	*Количество затронутых строк*	*Общий объем данных повторного выполнения*	*В среднем для строки*
Вставка одной строки	1	2679	2679
Вставка 10 строк с помощью одного оператора	10	22260	2226
Вставка 200 строк с помощью одного оператора	200	442784	2213
Вставка 200 строк по одной	200	464224	2321
Изменение одной строки	1	4228	4228
Изменение 10 строк с помощью одного оператора	10	42520	4252
Изменение 200 строк с помощью одного оператора	200	849600	4248
Изменение 200 строк по одной	200	849700	4248
Удаление одной строки	1	2236	2236
Удаление 10 строк с помощью одного оператора	10	23688	2369
Удаление 200 строк с помощью одного оператора	200	469152	2345
Удаление 200 строк по одной	200	469212	2346

Интересно отметить, что при изменении 200 строк с помощью одного оператора и по одной генерируется одинаковый объем данных повторного выполнения. Это верно и для операторов DELETE: один или 200 операторов — результат одинаков. Операторы вставки ведут себя немного иначе. При вставке строк по одной генерируется немого больше данных повторного выполнения, что вполне логично, если учесть, что при вставке по одной строке данные в блоках организуются немного не так, как при множественной вставке (при этом необходимо выполнить немного больше работы).

Как видите, объем генерируемых данных повторного выполнения зависит от объема изменяемых данных. При вставке строк размером 2000 байт генерируется немногим более 2000 байт для каждой строки. При выполнении оператора UPDATE объем генерируемых данных повторного выполнения удваивается (если помните, в журнал записываются данные и изменения в сегменте отката, так что это вполне логично). Оператор **DELETE** генерирует данные почти такого же объема, что и оператор **INSERT. В** сегменте отката записана вся строка, и это отражается в журнале, но записываются еще и изменение в блоке, что и объясняет небольшие расхождения. Поэтому, если известен объем изменяемых данных и то, **как** они будут изменяться, определить объем данных повторного выполнения легко.

Этот пример не показывает, что построчная обработка так же эффективна, как и обработка множества строк. Он показывает только, что при этом генерируется одинаковый объем данных повторного выполнения. В предыдущих главах было продемонстрировано, что процедурная обработка строк никогда не бывает настолько эффективной, как обработка множеств. Кроме того, если выполнять в цикле операторы **COMMIT,** как делают многие исходя из ошибочного предположения, что этим экономят ресурсы, проблема только усложняется. Теперь, зная, как оценить объем генерируемых данных повторного выполнения, можно четко показать последствия этой неудачной идеи. Используем ту же схему, что и в предыдущем примере, и посмотрим, что произойдет при выполнении оператора **COMMIT** в цикле:

```
tkyte@TKYTE816> declare
  2   l_redo_size   number;
  3     l_cnt        number  := 200;
  4     procedure  report
  5     is
  6     begin
  7         select  value-l_redo_size  into  l_redo_size  from  redo_size;
  8         dbms_output.put_line('redo  size  =  '  ||  l_redo_size ||
  9                               ' rows = ' || l_cnt || ' ' ||
 10                               to_char(l_redo_size/l_cnt, '99,999.9'} ||
 11                              '  bytes/row');
 12      end;
 13  begin
 14     select  value  into  l_redo_size  from  redo_size;
 15     fox x in  (select  object_id,  object_name,  created
 16                  from  all_objects
 17                 where  rownum <= l_cnt)
 18     loop
 19             insert  into  t  values
```

```
 20            (x.object_id, x.object_name, x.created);
 21            commit;
 22     end loop;
 23     report;
 24
 25     select value into l_redo_size from redo_size;
 26     for x in (select rowid rid from t)
 27     loop
 28         update t set y = lower(y) where rowid = x.rid;
 29         commit;
 30         end loop;
 31     report;
 32
 33     select value into l_redo_size from redo_size;
 34     for x in (select rowid rid from t)
 35     loop
 36         delete from t where rowid = x.rid;
 37         commit;
 38     end loop;
 39     report;
 40  end;
 41  /
redo size = 530396 rows = 200    2,652.0 bytes/row
redo size = 956660 rows = 200    4,783.3 bytes/row
redo size = 537132 rows = 200    2,685.7 bytes/row

PL/SQL procedure successfully completed.
```

Как видите, фиксация каждой строки заметно увеличила объем генерируемых данных повторного выполнения (итоговые результаты представлены в следующей таблице). Есть и другие причины, связанные с производительностью, объясняющие, почему не стоит фиксировать каждую измененную строку (или группу строк). Мы видели непосредственное доказательство этого в примере выше, где фиксация каждой строки требует в три раза больше времени, чем фиксация после завершения транзакции. Дополнительные затраты времени связаны с обращением к ядру сервера, установкой дополнительных защелок и возникновением конфликтов при доступе к общим ресурсам при выполнении каждого оператора. Итак, если что-то можно сделать с помощью одного оператора SQL — делайте это именно так. Кроме того, фиксируйте транзакцию, только когда она закончена, не раньше.

Операция	***Количество затронутых строк***	***Общий объем данных повторного выполнения*** *(без фиксации)*	***Общий объем данных повторного выполнения*** *(с фиксацией)*	***%увеличения***
Вставка 200 строк	200	442784	530396	20%
Изменение 200 строк	200	849600	956660	13%
Удаление 200 строк	200	469152	537132	14%

Представленный выше метод пригодится и для оценки побочных эффектов различных вариантов работы. Один из часто возникающих вопросов связан с триггерами. Помимо того, что в триггере BEFORE можно изменять указанные значения строки, есть ли еще какие-то особенности применения триггеров? Как оказалось, есть. Триггер BEFORE создает дополнительные данные повторного выполнения, даже не изменив ни одного значения в строке. Этот вопрос достоин отдельного изучения, и с помощью представленных выше методов можно убедиться, что:

- триггер BEFORE или AFTER не влияет на объем данных повторного выполнения для операторов DELETE;
- оператор INSERT генерирует дополнительно одинаковый объем данных повторного выполнения при наличии триггера BEFORE или AFTER;
- на объем данных повторного выполнения, генерируемых оператором UPDATE, влияет только наличие триггера BEFORE — триггер AFTER не генерирует дополнительных данных;
- размер строки влияет на объем дополнительно генерируемых данных повторного выполнения для операторов INSERT, но не для операторов UPDATE.

Используем таблицу T из предыдущего примера:

```
create table t (x int, y char(N), z date);
```

но создадим несколько ее версий с разными значениями N. В данном случае мы используем значения N = 30, 100, 500, 1000 и 2000 для получения строк разного размера. Я использовал простую таблицу для регистрации результатов нескольких экспериментов:

```
create table log (what varchar2(15), — будет иметь значения по trigger, after
                                     — или before
                  op varchar2(10),   — будет иметь значения insert, update или
                                     — delete
                  rowsize int,       — будет содержать размер столбца Y
                  redo_size int,     — будет содержать объем сгенерированных
                                     — данных повторного выполнения
                  rowcnt int);       — будет содержать количество обработанных
                                     — строк
```

Выполнив тест с различными размерами столбца Y, проанализируем результаты. Я использую следующую хранимую процедуру для генерации транзакций и записи объема сгенерированных данных повторного выполнения. Подпроцедура REPORT — это локальная процедура (видимая только в процедуре DO_WORK), выдающая на экран информацию о происходящем и записывающая полученные результаты в таблицу LOG. Основное тело процедуры выполняет проверяемые транзакции. Все начинается с запоминания текущего объема данных повторного выполнения в текущем сеансе, выполнения определенных действий, а затем генерируется отчет:

```
tkyte@TKYTE816> create or replace procedure do_work(p_what in varchar2)
  2  as
  3      l_redo_size number;
  4      l cnt        number := 200;
```

```
 5
 6    procedure report(l_op in varchar2)
 7    is
 8    begin
 9        select value-l_redo_size into l_redo_size from redo_size;
10      dbms_output.put_line(l_op ||  ' redo size = ' || l_redo_size ||
11                           ' rows' = '  || l_cnt ||  '  '  ||
12                             to_char(l_redo_size/l_cnt,"99,999.9')  ||
13                              '  bytes/row');
14      insert into log
15      select p_what, l_op, data_length, l_redo_size, l_cnt
16        from user_tab_columns
17       where table_name =  'T'
18         and column_name =  'Y';
19     end;
20  begin
21    select value into l_redo_size from redo_size;
22    insert into t
23    select object_id, object_name, created
24      from all_objects
25     where rownum <= l_cnt;
26    l_cnt := sql%rowcount;
27    commit;
28    report('insert');
29
30     select value into l_redo_size from redo_size;
31    update t set y=lower(y);
32    l_cnt := sql%rowcount;
33    commit;
34   report('update');
35
36    select value into l_redo_size from redo_size;
37    delete from t;
38   l_cnt := sql%rowcount;
39    commit;
40   report('delete');
41  end;
42  /

Procedure created.
```

Теперь, когда все готово, я удалю **и** создам таблицу **T, изменяя размер** столбца Y. Затем выполню следующий код для тестирования различных сценариев (без триггеров, с триггером **BEFORE** и с триггером **AFTER):**

```
tkyte@TKYTE816> truncate table t;

Table truncated.

tkyte@TKYTE816> exec do_work('no trigger');
insert redo size = 443280 rows = 200    2,216.4 bytes/row
update redo size = 853968 rows = 200    4,269.8 bytes/row
delete redo size = 473620 rows - 200    2,368.1 bytes/row
```

```
PL/SQL procedure successfully completed.

tkyte@TKYTE816> create or replace trigger before_insert_update_delete
  2  before insert or update or delete on T for each row
  3  begin
  4            null;
  5  end;
  6  /

Trigger created.

tkyte@TKYTE816> truncate table t;

Table truncated.

tkyte@TKYTE816> exec do_work('before trigger');
insert redo size = 465640 rows = 200    2,328.2 bytes/row
update redo size = 891628 rows = 200    4,458.1 bytes/row
delete redo size = 473520 rows = 200    2,367.6 bytes/row

PL/SQL procedure successfully completed.

tkyte@TKYTE816> drop trigger before_insert_update_delete;

Trigger dropped.

tkyte@TKYTE816> create or replace trigger after_insert_update_delete
  2  after insert or update or delete on T
  3  for each row
  4  begin
  5            null;
  6  end;
  7  /

Trigger created.

tkyte@TKYTE816> truncate table t;

Table truncated.

tkyte@TKYTE816> exec do_work('after trigger');
insert redo size = 465600 rows = 200    2,328.0 bytes/row
update redo size = 854028 rows = 200    4,270.1 bytes/row
delete redo size = 473580 rows = 200    2,367.9 bytes/row

PL/SQL procedure successfully completed.
```

Выше представлен результат теста, в котором столбец Y имел размер 2000 байт. После завершения всех тестов я получил следующие результаты запроса к регистрационной таблице:

```
tkyte@TKYTE816> break on op skip 1
tkyte@TKYTE816> set numformat 999,999
```

```
tkyte@TKYTE816> select op, rowsize, no_trig, before_trig-no_trig,
after_trig-no_trig
  2  from ( select op, rowsize,
  3             sum(decode(what, 'no trigger', redo_size/rowcnt,0))
                                                          no_trig,
  4             sum(decode(what, 'before trigger', redo_size/rowcnt, 0))
                                                          before_trig,
  5             sum(decode(what, 'after trigger', redo_size/rowcnt, 0))
                                                          after_trig
  6            from log
  7          group by op, rowsize
  8       )
  9   order by op, rowsize
 10  /

OP        ROWSIZE  NO_TRIG  BEFORE_TRIG-NO_TRIG  AFTER_TRIG-NO_TRIG

delete         30      272                    0                   0
              100      344                   -0                  -0
              500      765                   -0                  -0
            1,000    1,293                   -1                  -0
            2,000    2,368                   -1                  -0

insert         30       60                  213                 213
              100      136                  208                 208
              500      574                  184                 184
            1,000    1,113                  162                 162
            2,000    2,216                  112                 112

update         30      294                  189                   0
              100      431                  188                   0
              500    1,238                  188                  -0
            1,000    2,246                  188                  -0
            2,000    4,270                  188                   0

15 rows selected.
```

Если вам непонятен этот запрос или способ группировки результирующего множества, обратитесь к главе 12, посвященной аналитическим функциям, в которой я детально описываю способы группировки результирующего множества.

Внутренний запрос сгенерировал результирующее множество, содержащее средний объем в байтах данных повторного выполнения для строки для каждого из трех тестов. Внешний запрос просто выдает среднее количество байтов для строки при отсутствии триггера, а последующие два столбца показывают отличие других случаев от тестов при отсутствии триггера. Итак, для тестов с операторами DELETE заметного различия в объеме генерируемых для каждой строки данных повторного выполнения нет. Для операторов INSERT, однако, можно обнаружить дополнительные расходы ресурсов — от 213 до 112 байтов на строку, независимо от типа использованного триггера (BEFORE или

AFTER). Чем больше строка, тем меньше почему-то эти дополнительные расходы (я не разбирался, **почему** это так, но именно так и получается). Наконец, для операторов **UPDATE** обнаруживается два факта. Во-первых, объем генерируемых данных повторного выполнения не зависит от размера строки — дополнительные расходы на выполнение операторов **UPDATE** постоянны. Во-вторых, триггер **AFTER** оказывается намного эффективнее для операторов **UPDATE,** поскольку он вообще не влияет на генерацию данных повторного выполнения. Поэтому можно использовать эмпирическое правило: для операторов **UPDATE** по возможности используйте триггеры **AFTER.** Триггер **BEFORE** имеет смысл использовать, только если требуются его специфические функциональные возможности.

Итак, теперь вы знаете, как оценить объем данных повторного выполнения, что необходимо делать всем разработчикам. Для этого требуется:

- оценить размер транзакции (сколько данных изменяется);
- добавить от 10 до 20 процентов дополнительного пространства, в зависимости от количества изменяемых строк (чем больше строк, тем меньше процент дополнительных расходов);
- удвоить полученное значение для операторов **UPDATE.**

В большинстве случаев это дает вполне приемлемые результаты для оценки. Удвоение объема для операторов **UPDATE** — это упрощение; реальный объем зависит от способа изменения данных. Удвоение предполагает, что берется строка размером X байтов, и в результате изменения получится строка тоже размером X байтов. Если увеличивается маленькая строка, значение не удваивается (результат ближе к тому, который получается при выполнении оператора **INSERT).** Если уменьшается большая строка, значение тоже не удваивается (результат сходен с тем, который получается при выполнении оператора **DELETE).** Удвоение — "худший случай", поскольку некоторые опции и средства влияют на объем генерируемых данных, например наличие (или отсутствие, как в моем случае) индексов. Объем работы, необходимой для поддержки индекса, для разных операторов **UPDATE** может отличаться. Побочные эффекты триггеров также необходимо учесть (помимо фиксированных затрат ресурсов, описанных выше). Некоторые операции, выполняемые от имени сеанса неявно, например при установке опции **ON DELETE CASCADE,** также необходимо учитывать. Это поможет оценить объем данных повторного выполнения для оценки требований к пространству/производительности. Только тестирование в реальных условиях даст гарантированный результат. На примере представленного выше сценария вы сможете понять, как определить этот объем для своих объектов и транзакций.

Можно ли отключить генерацию записей в журнал повторного выполнения?

Этот вопрос задают очень часто. Простой и короткий ответ — "нет". Журнализация данных повторного выполнения принципиально важна для базы данных и расходы на нее нельзя считать потерей ресурсов. Журнализация действительно необходима, как бы

вы к ней не относились. Это факт, именно так работает сервер. Однако есть ряд операций, которые иногда можно выполнять, не генерируя данных в журнал повторного выполнения.

Некоторые SQL-операторы поддерживают конструкцию **NOLOGGING.** Это не означает, что все операции с объектом будут выполняться без генерирования данных в журнал повторного выполнения. Просто некоторые специфические операции будут генерировать **намного меньше** данных повторного выполнения, чем обычно. Обратите внимание: "будут генерировать намного меньше", а не "вообще не будут генерировать". Все операции генерируют определенный объем данных повторного выполнения, поскольку все операции со словарем данных журнализируются, независимо от режима. Объем генерируемых данных повторного выполнения может быть намного меньше. Например, я выполнил следующие действия в базе данных, работающей в режиме **ARCHIVELOG.** Если выполнить их в режиме **NOARCHIVELOG,** вы не увидите разницы. Оператор CREATE **TABLE** в базе данных, работающей в режиме **NOARCHIVELOG,** не будет журнализироваться, за исключением изменений в словаре данных. Пользователи сервера Oracle версии 7.3, однако, увидят различия, поскольку такая оптимизация в этой версии сервера еще не выполнялась. Им также придется использовать ключевое слово **UNRECOVERABLE** вместо **NOLOGGING.** В прежних версиях Oracle **NOLOGGING** (без журнализации) означало **UNRECOVERABLE** (невосстановимо). Если хочется увидеть отличие при работе базы данных в режиме **NOARCHIVELOG,** замените операторы **DROP TABLE** и **CREATE TABLE** операторами **DROP INDEX** и **CREATE INDEX** для какой-либо таблицы. Эти операторы всегда журнализируются, независимо от режима работы сервера. Отсюда и ценный совет: тестируйте систему в том режиме, в котором она будет реально работать, поскольку ее поведение может зависеть от режима. Производственная система должна работать в режиме **ARCHIVELOG.** Если выполняется много операций, которые в этом режиме генерируют данные повторного выполнения, а в режиме **NOARCHIVELOG** — нет, то это лучше выяснить в ходе тестирования, а не при вводе в промышленную эксплуатацию! Теперь приведу пример использования конструкции **NOLOGGING:**

```
tkyte@TKYTE816> column value new_value old_value
tkyte@TKYTE816> select value from redo_size;
    VALUE

  5195512

tkyte@TKYTE816> create table t
  2  as
  3  select * from all_objects
  4  /

Table created.

tkyte@TKYTE816> select value-&old_value REDO_GENERATED from redo_size;
old   1: select value-Sold_value REDO_GENERATED from redo_size
new   1: select value-   5195512 REDO_GENERATED from redo_size
```

```
REDO_GENERATED

      2515860
```

В моей базе данных сгенерировано более 2,5 Мбайт данных повторного выполнения.

```
tkyte@TKYTE816> drop table t;

Table dropped.

tkyte@TKYTE816> select value from redo_size;

    VALUE

  7741248

tkyte@TKYTE816> create table t
  2  NOLOGGING
  3  as
  4  select * from all_objects
  5  /

Table created.

tkyte@TKYTE816> select value-Sold_value REDO_GENERATED from redo_size;
old   1: select value-&old_value REDO_GENERATED from redo_size
new   1: select value-  7741248  REDO_GENERATED from redo_size

   REDO_GENERATED

            43264
```

А в этот раз сгенерировано только около 50 Кбайт данных повторного выполнения.

Как видите, различие существенное: 2,5 Мбайт данных повторного выполнения или 50 Кбайт. 2,5 Мбайт — это данные таблицы; они были записаны непосредственно на диск без генерирования данных повторного выполнения. Теперь, конечно, стало понятно, что все операции, для которых это возможно, надо выполнять с опцией NOLOGGING, не так ли? На самом деле, **нет.** Ее необходимо использовать очень осторожно и только после предварительного согласования с ответственным за резервное копирование и восстановление. Пусть частью приложения является таблица, созданная именно так (например, оператор **CREATE TABLE AS SELECT NOLOGGING** является частью сценария установки новой версии). Пользователи будут изменять эту таблицу в течение дня. А ночью произойдет сбой диска, на котором находится таблица. "Никаких проблем", — скажет администратор базы данных, — мы же работаем в режиме **ARCHIVELOG,** так что можем восстановить данные". Проблема, однако, в том, что таблица, созданная как нежурнализируемая, не может быть восстановлена из архивного журнала повторного выполнения. Эта таблица не восстановима, и именно в этом основная особенность операций в режиме **NOLOGGING** — их использование необходимо согласовать с администратором базы данных, отвечающим за систему в целом. Если вы используете их, а другие пользователи об этом не знают, это сделает невозможным

восстановление базы данных в случае сбоя носителя. Такие операции надо использовать разумно и осторожно.

Про операции, выполняемые с опцией **NOLOGGING,** важно знать следующее.

- Определенный объем данных повторного выполнения на самом деле будет генерироваться. Эти данные обеспечивают защиту словаря данных. Избежать их генерации нельзя. Объем генерируемых данных будет значительно меньше, но генерироваться они все равно будут.
- Установка опции **NOLOGGING** не предотвращает генерирования данных повторного выполнения последующими операторами. В представленном выше примере не была создана таблица, действия с которой не регистрируются в журнале. Все последующие "обычные" действия, выполняемые операторами **INSERT, UPDATE** и **DELETE,** будут записываться в журнал. Другие специальные действия, например непосредственная загрузка с помощью утилиты **SQLLDR** или непосредственные вставки с помощью операторов **INSERT** /*+ **APPEND** */, регистрироваться в журнале не будут. В общем случае, однако, действия, выполняемые приложениями с этой таблицей, регистрируются в журнале повторного выполнения.
- После выполнения действий с опцией **NOLOGGING** в базе данных, работающей в режиме **ARCHIVELOG,** необходимо как можно быстрее создать базовую резервную копию затронутых файлов данных. Это необходимо для предотвращения потери последующих изменений соответствующих объектов при сбое носителя. Мы не потеряем сами изменения, поскольку они записываются в журнал повторного выполнения. Будут потеряны данные, к которым эти изменения относятся.

Есть два способа использования опции **NOLOGGING.** Один способ уже продемонстрирован: добавить ключевое слово **NOLOGGING** в соответствующем месте SQL-оператора. Другой способ позволяет неявно выполнять действия в режиме **NOLOGGING.** Например, можно изменить индекс так, чтобы он по умолчанию работал в режиме **NOLOGGING.** Это означает, что при последующем выполнении непосредственных загрузок или непосредственных вставок, затрагивающих этот индекс, изменения не будут регистрироваться (при изменении индекса не будут генерироваться данные повторного выполнения; для других индексов и самой таблицы — возможно, но для этого индекса — нет).

В режиме **NOLOGGING** возможны следующие действия:

- создание и изменение (перестройка) индексов;
- множественные "непосредственные" вставки с помощью подсказки /*+ **APPEND** */;
- действия с большими объектами (изменения больших объектов регистрировать в журнале необязательно);
- создание таблиц с помощью операторов **CREATE TABLE AS SELECT;**
- изменение таблиц с помощью **ALTER TABLE,** такие как **MOVE** и **SPLIT;**

- выполнение оператора **TRUNCATE** (но для него указывать конструкцию **NOLOGGING** не надо, поскольку он всегда выполняется в режиме NOLOGGING).

При правильном использовании в базе данных, работающей в режиме **ARCHIVELOG,** опция **NOLOGGING** может ускорить выполнение многих действий, существенно уменьшая объем данных, генерируемых в журнал повторного выполнения. Предположим, необходимо перенести таблицу из одного табличного пространства в другое. Эту операцию можно выполнить непосредственно перед резервным копированием: изменить таблицу с помощью оператора **ALTER TABLE,** переведя ее в режим **NOLOGGING,** перенести ее, пересоздать индексы (тоже без журнализации), а затем снова перевести таблицу в режим журнализации изменений с помощью оператора **ALTER TABLE.** В результате этого действие, ранее требовавшее X часов, может выполняться, скажем, X/2 часов. Правильное использование этой возможности включает согласование с администратором базы данных или ответственным за резервное копирование и восстановление. Если они не знают о ее использовании и произойдет сбой носителя, данные могут быть потеряны. Это надо учитывать.

Не удается выделить новый журнал?

Я постоянно наблюдаю это, хотя и не на своем сервере, конечно. Вы получаете предупреждения об этом (их можно найти в файле журнала сообщений, **alert.log,** сервера):

```
Sun Feb 25 10:59:55 2001
Thread 1 cannot allocate new log, sequence 326
Checkpoint not complete
```

В журнале может оказаться сообщение **Archival required,** а не **Checkpoint not complete,** но результат — практически тот же. Именно за появлением таких сообщений должен следить администратор базы данных. Если вдруг он этого не делает, вы должны искать их сами. Такие сообщения будут записываться в файл **alert.log** на сервере при каждой попытке сервера повторно использовать активный журнал повторного выполнения, когда оказывается, что этого делать нельзя. Это будет происходить, когда процесс **DBWR** еще не закончил обработку контрольной точки для данных, защищаемых этим журналом повторного выполнения, или когда процесс **ARCH** не закончил копирование файла журнала повторного выполнения в архив. Если названия процессов **DBWR** и **ARCH** ничего для вас не значат, почитайте о них подробнее в главе 2. В этот момент сервер фактически **останавливается** для пользователей. Процесс **DBWR** или **ARCH** получает приоритет для сброса блоков на диск. После завершения обработки контрольной точки или архивирования нормальная работа восстанавливается. Причина приостановки выполнения запросов пользователей сервером в том, что нет места для записи выполняемых ими изменений. Сервер Oracle пытается повторно использовать активный файл журнала повторного выполнения. Но поскольку этот файл либо необходим для восстановления базы данных **(Checkpoint not complete),** либо еще не скопирован в архив **(Archival required),** серверу Oracle придется подождать (и конечным пользователям тоже), пока файл журнала повторного выполнения будет безопасно использовать.

Если оказывается, что сеансы долго ждут событий **'log file switch', 'log buffer space'** или **'log file switch checkpoint or archival incomplete',** скорее всего вы столкнулись именно с этой проблемой (о том, как узнать, каких событий ожидает сеанс, см. в главе 10). С ней можно столкнуться при продолжительных изменениях базы данных, если файлы журнала повторного выполнения имеют несоответствующий размер или когда требуется настройка работы процессов **DBWR** и **ARCH** администратором базы данных или системным администратором. Я часто сталкивался с этой проблемой в стандартной ("starter") базе данных, которую не настраивали. В стандартной базе данных, копируемой при установке с дистрибутивного носителя, журналы повторного выполнения обычно слишком малы для поддержки выполнения существенных действий (включая первоначальное построение словаря данных). При загрузке данных в базу оказывается, что первые 1000 строк вставляются быстро, затем работа продолжается медленно; 1000 строк вставляются быстро, затем сервер зависает, затем опять работает быстро и снова зависает, и т.д. Это признаки описанной выше ситуации.

Для решения этой проблемы можно сделать следующее.

- Ускорить работу процесса **DBWR.** Пусть администратор базы данных настроит процесс **DBWR:** включит поддержку асинхронного ввода/вывода, используя подчиненные процессы ввода/вывода **DBWR** или запустив несколько процессов **DBWR.** Посмотрите на статистическую информацию о вводе/выводе в системе — может обнаружиться "горячий" диск или ряд дисков, ввод/вывод на которые надо распределить. Такой же общий совет относится и к процессу **ARCH.** Преимущество этого решения в том, что оно "бесплатное" — повышение производительности достигается без изменения логики/структур/кода. Недостатков у него по сути нет.

- Добавить файлы в журнал повторного выполнения. Это отсрочит возникновение сообщений **Checkpoint not complete,** причем, после добавления определенного количества — настолько, что они вообще не будут выдаваться (у процесса **DBWR** появляется пространство для обработки контрольной точки). Это же относится и к сообщению **Archival required.** Преимущество этого подхода — устранение "пауз" в работе системы. Недостаток — требуется больше дискового пространства, но преимущество в данном случае намного перевешивает недостаток.

- Пересоздать файлы журнала, сделав их больше. Это отсрочит заполнение активного файла журнала повторного выполнения и увеличит период времени, после которого он снова понадобится. Это решает и проблему с выдачей сообщений **Archival required,** если файлы журнала повторного выполнения используются в "пульсирующем" режиме. Если за периодами интенсивного генерирования данных повторного выполнения (еженощные загрузки данных, пакетная обработка) следуют периоды относительного затишья, увеличив активные журналы повторного выполнения, можно дать процессу **ARCH** дополнительное время для работы в периоды затишья. Преимущества и недостатки этого подхода те же, что и в случае добавления файлов. Кроме того, может быть отсрочена обработка контрольной точки, поскольку она выполняется при переходе с одного логического журнала на другой, а переключения будут происходить реже.

- Вызвать более частую и постоянную обработку контрольной точки. Для этого можно уменьшить буферный кэш (что вообще-то нежелательно) или изменить значения некоторых параметров инициализации, в частности **FAST_START_IO_TARGET, DB_BLOCK_MAX_DIRTY_TARGET, LOG_CHECKPOINT_INTERVAL** и **LOG_CHECKPOINT_TIMEOUT.** Это приведет к более частому сбросу грязных блоков на диск процессом **DBWR.** К преимуществам этого подхода можно отнести уменьшение времени восстановления в случае сбоя. Будет оставаться меньше изменений в активных журналах повторного выполнения для сброса на диск. Недостаток же в том, что блоки будут записываться на диск чаще. Буферный кэш будет работать с меньшей эффективностью, да и весь описанный далее механизм очистки блоков может при этом пострадать.

Выбор подхода зависит от обстоятельств. Эту проблему надо решать на уровне всей базы данных, с учетом работы экземпляра в целом.

Очистка блоков

Вспомните, в главе 3 я объяснял, что блокировки фактически являются атрибутами данных и хранятся в заголовке блока. Побочный эффект этого в том, что при следующем обращении к блоку может понадобиться очистить его, т.е. удалить информацию о транзакциях. При этом генерируются данные повторного выполнения, и блок становится "грязным", если он таковым еще не был. Это означает, что простой оператор SELECT может генерировать данные повторного выполнения и вызывать запись на диск множества блоков при следующей обработке контрольной точки. В большинстве случаев, однако, этого не происходит. Если транзакции — маленького и среднего размера (OOT) или выполняется анализ таблиц после множественных операций в хранилище данных, блоки очищаются автоматически. Как было описано ранее, в разделе "Что происходит при фиксации?", при фиксации транзакции происходит повторное обращение к блокам, находящимся в области SGA, если они доступны (другой сеанс их не изменяет), и их очистка. Это действие называют *очисткой при фиксации.* При фиксации транзакции блоки очищаются настолько, чтобы при последующем выполнении оператора **SELECT** (чтении) их очищать не пришлось. Только при изменении блока придется удалять находящуюся в нем информацию о транзакции, а поскольку данные повторного выполнения при этом и так генерируются, эта очистка проходит незаметно.

Можно принудительно вызвать очистку, чтобы увидеть ее побочные эффекты, если понимать, как выполняется очистка при обработке контрольной точки. Сервер Oracle выделяет списки измененных блоков в списке фиксации, связанном с транзакцией. Каждый их этих списков отслеживает 20 блоков, и сервер Oracle будет выделять столько списков, сколько необходимо. Если суммарный объем измененных транзакцией блоков превысит 10 процентов буферного кэша, сервер Oracle прекратит выделение новых списков. Например, если буферный кэш имеет размер 3000 блоков, сервер Oracle будет автоматически поддерживать список размером до 300 блоков (10 процентов от 3000). При фиксации сервер Oracle будет обрабатывать каждый из выделенных списков по 20 указателей на блоки и, если блок доступен, быстро очистит его. Поэтому, если количество

измененных блоков не превышает 10 процентов буферного кэша и эти блоки доступны, сервер Oracle очистит их при фиксации транзакции. В противном случае блоки будут просто пропущены (не очищены). Зная это, можно создать искусственные условия для демонстрации того, как происходит очистка. Я установил параметру DB_BLOCK_BUFFERS небольшое значение — 300. Затем создал таблицу, строка которой занимает блок (т.е. двух строк ни в одном блоке не будет). Затем я вставил в эту таблицу 499 строк и выполнил оператор COMMIT. После этого измерил объем сгенерированных данных повторного выполнения, выполнил оператор SELECT, который обращается к каждому измененному блоку, и измерил объем данных повторного выполнения, сгенерированных оператором SELECT.

К удивлению многих, оператор SELECT действительно генерирует в этом случае данные повторного выполнения. Более того, измененные им блоки становятся "грязными" и снова сбрасываются на диск процессом DBWR. Это происходит вследствие очистки блоков. Затем я выполнил оператор SELECT еще раз, и он больше не сгенерировал данные повторного выполнения (что вполне предсказуемо, поскольку в этот момент блоки — "чистые").

```
tkyte@TKYTE816> create table t
  2   (x char(2000) default 'x',
  3    y char(2000) default 'y',
  4    z char(2000) default 'z')
  5  /

Table created.

tkyte@TKYTE816> insert into t
  2  select 'x','y','z'
  3   from all_objects where rownum < 500
  4  /

499 rows created.

tkyte@TKYTE816>  commit;

Commit complete.
```

Итак, вот таблица с одной строкой в каждом блоке (при размере блока базы данных 8 Кбайт). Давайте определим объем данных повторного выполнения, сгенерированных при чтении данных:

```
tkyte@TKYTE816> column value new_value old_value
tkyte@TKYTE816> select * from redo_size;

     VALUE

   3250592
tkyte@TKYTE816> select *
  2     from t
  3    where x = y;

no rows selected
```

```
tkyte@TKYTE816> select value-&old_value  REDO_GENERATED from redo_size;
old   1: select value-&old_value  REDO_GENERATED from redo_size
new   1: select value-   3250592   REDO_GENERATED from redo_size

   REDO_GENERATED

            29940

tkyte@TKYTE816> commit;

Commit complete.
```

Итак, при выполнении оператора SELECT было сгенерировано около 30 Кбайт данных повторного выполнения. Она представляет заголовки блоков, измененные при полном просмотре таблицы T. Процесс DBWR в дальнейшем запишет эти измененные блоки на диск. Если теперь выполнить запрос повторно:

```
tkyte@TKYTE816> select value from redo_size;

    VALUE

  3280532

tkyte@TKYTE816> select *
  2     from t
  3    where x = y;

no rows selected

tkyte@TKYTE816>
tkyte@TKYTE816> select value-&old_value  REDO_GENERATED from redo_size;
old   1: select value-&old_value  REDO_GENERATED from redo_size

new   1: select value-   3280532   REDO_GENERATED from redo_size

   REDO_GENERATED

                0

Commit complete.
```

то можно убедиться, что данные повторного выполнения не генерируются; все блоки — чистые.

Если снова выполнить этот пример при размере буферного кэша не 300 блоков, а 6000, окажется, что при выполнении обоих операторов SELECT данные повторного выполнения не генерируются: ни один из блоков при выполнении этих операторов не изменился. Так происходит потому, что 499 измененных блоков составляют менее 10 процентов буферного кэша, а наш сеанс — единственный. Ни один другой сеанс не работает с данными, никто не вызывает сброс данных на диск и не обращается к измененным блокам. В реальных системах блоки иногда не очищаются, и это вполне нормально.

Чаще всего подобная ситуация встречается при вставке (как было показано выше), изменении или удалении большого количества строк, когда затрагивается много блоков

базы данных (при изменении данных объемом более 10 процентов буферного кэша она, определенно, возникнет). Первый же запрос, выполненный к затронутому блоку после этого, сгенерирует немного данных повторного выполнения, сделает блок "грязным" и, кроме того, может вызвать его перезапись, если процесс уже успел сбросить блок на диск или экземпляр был остановлен, что вызвало очистку всего буферного кэша. С этим ничего нельзя поделать. Это нормально и вполне предсказуемо. Если бы сервер Oracle не выполнял такую отложенную очистку блоков, для выполнения оператора COMMIT могло бы потребоваться столько же времени, как и для выполнения самой транзакции. При фиксации пришлось бы повторно обработать каждый затронутый транзакцией блок, возможно, повторно считав его при этом с диска (блок уже мог быть сброшен). Если не учитывать очистку блоков и способ ее выполнения, некоторые события будут казаться загадочными, происходящими без видимой причины. Например, изменен большой объем данных и транзакция зафиксирована. После этого выполняется запрос для проверки результатов. Этот запрос почему-то записывает огромные объемы данных на диск и генерирует большое количество данных повторного выполнения. Это кажется невозможным, если не знать об очистке блоков. Вы пытаетесь кому-то показать это, но ситуация невоспроизводима, поскольку блоки при повторном запросе — "чистые". После этого вы относите ситуацию к разряду необъяснимых "загадок".

В системах OOT с такой ситуацией почти никогда не сталкиваются. Все транзакции — короткие и простые. Изменяется несколько блоков, и они тут же очищаются. В хранилище данных, где выполняются множественные изменения данных после загрузки, очистку блоков необходимо учитывать при проектировании. Некоторые операции будут создавать данные в "чистых" блоках. Например, при выполнении оператора CREATE TABLE AS SELECT, непосредственной загрузке и непосредственной вставке данных создаются "чистые" блоки. Операторы UPDATE, обычные операторы INSERT или DELETE могут создавать блоки, которые придется очищать при первом чтении. Это может повлиять на работу, если выполняются следующие операции:

- множественная загрузка новых данных в хранилище данных;
- выполнение изменений всех данных сразу после загрузки (в результате чего создаются блоки, требующие очистки);
- при этом пользователи запрашивают данные.

Необходимо учитывать, что первый запрос, обращающийся к данным, потребует дополнительной обработки, если придется очищать блоки. С учетом этого можно самостоятельно "затронуть" данные сразу после изменения. Только что загружен или изменен большой объем данных; их надо проанализировать. Возможно, придется создать ряд отчетов, проверяющих выполнение загрузки. В результате блоки будут очищены, и следующем запросу не придется это делать. Еще лучше сразу после множественной загрузки данных обновить статистическую информацию — все равно это придется сделать. Выполнение оператора ANALYZE для обновления статистической информации тоже очистит все блоки.

Конфликты при доступе к журналу

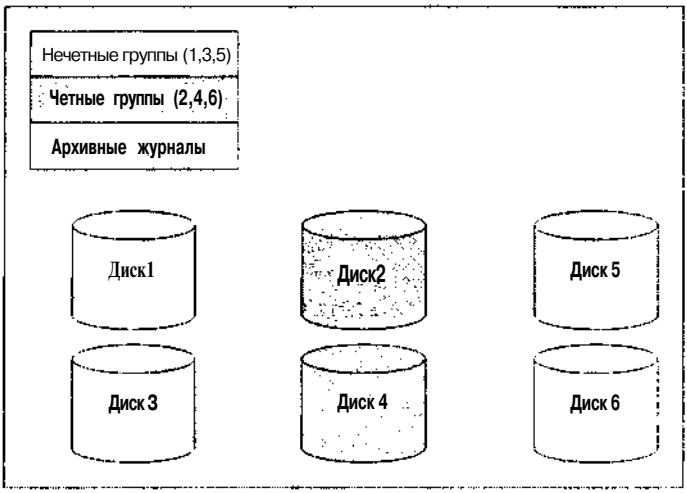
Эти конфликты, как и сообщения **Cannot allocate new log,** должен устранять администратор базы данных, как правило, совместно с системным администратором. Однако заметить их может и разработчик, если администраторы следят за сервером недостаточно внимательно. При описании важных представлений динамической производительности V$ в главе 10 мы разберемся, как понять, чего именно ждет сеанс. Часто дольше всего сеансу приходится ждать события **'log file sync'.** Это значит, что возникают конфликты при доступе к журналам повторного выполнения; работа с ними выполняется недостаточно быстро. Это может происходить по многим причинам. Одна из причин, связанных с приложениями (ее не может устранить администратор базы данных — это должен делать разработчик), — слишком частая фиксация, например фиксация в цикле после выполнения каждого оператора **INSERT.** Здесь фиксация выполняется искусственно, исходя из ложного предположения, что так можно сэкономить ресурсы. Слишком частая фиксация, помимо того, что это плохая практика программирования, — еще и гарантированный способ добавить множество ожиданий доступа к журнальным файлам, поскольку придется ждать, пока процесс **LGWR** сбросит буферы журнала повторного выполнения на диск. Обычно процесс **LGWR** может это делать в фоновом режиме, и ожидать не придется. При фиксации транзакции также приходится ожидать чаще и дольше, чем это необходимо. Предполагая, что все транзакции имеют правильный размер (фиксация выполняется не чаще, чем это диктуется выполняемыми действиями), основными причинами, вызывающими ожидание доступа к файлам журнала повторного выполнения являются следующие.

- Файлы журнала повторного выполнения размещены на низкоскоростном устройстве. Диск просто имеет низкую производительность. Пришло время покупать более высокоскоростные диски.
- Размещение журнала повторного выполнения на том же устройстве, что и другие файлы. Журнал повторного выполнения создавался для выполнения больших последовательных записей и размещения на отдельных устройствах. Если другие компоненты системы, даже компоненты сервера Oracle, пытаются читать и записывать данные на это устройство одновременно с процессом **LGWR,** определенный уровень конфликтов неизбежен. Необходимо обеспечить исключительный доступ процесса **LGWR** к соответствующим устройствам.
- Использование устройств с буферизацией. Речь идет об использовании файлов в файловой системе (вместо неформатированных дисков). Операционная система буферизует данные, СУБД тоже буферизует данные (речь идет о буфере журнала повторного выполнения). Двойная буферизация замедляет работу. По возможности используйте неформатированные устройства. Способ такого использования зависит от операционной системы и устройства, но обычно это возможно.
- Размещение журнала повторного устройства на низкоскоростных дисковых массивах, например RAID-5. Массивы RAID-5 прекрасно подходят для чтения, но резко снижают производительность записи. Как было показано ранее, при фик-

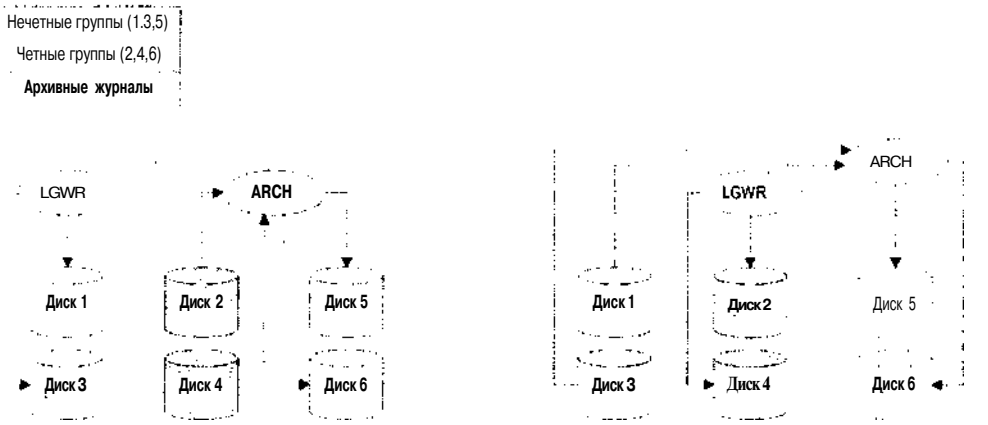
сации транзакции необходимо дождаться, пока процесс LGWR сбросит данные на диск. Использование любых технологий, замедляющих процесс записи, — плохая идея.

Желательно использовать для журнализации пять отдельных устройств, оптимально — шесть, чтобы можно было выполнять резервное копирование архивов. Сегодня, при наличии дисков объемом 20, 36 и более Гбайт, это становится все сложнее, но если выделить четыре высокоскоростных диска небольшой емкости и один или два диска большей емкости, можно существенно ускорить работу процессов LGWR и ARCH. Эти диски разбиваются на три группы:

- группа журналов повторного выполнения 1 — диски 1 и 3
- группа журналов повторного выполнения 2 — диски 2 и 4
- архив — диск 5 и, необязательно, диск 6 (диск большого объема)



Группу журналов повторного выполнения 1 с членами А и Б помещаете в группу 1. Группу журналов повторного выполнения 2 с членами В и Г помещаете в группу 2. Если есть группы 3, 4 и т.д., они размещаются, соответственно, на нечетной и четной группе дисков. При использовании группы 1 процесс LGWR будет записывать на диск 1 и диск 3 одновременно. При заполнении этой группы процесс LGWR перейдет к дискам 2 и 4. Когда эти диски заполнятся, он вернется к дискам 1 и 3. Тем временем процесс ARCH будет обрабатывать заполненные активные журналы повторного выполнения и записывать их в группу 3 на диск большой емкости. В результате ни процесс ARCH, ни процесс LGWR никогда не читает диск, на который идет запись, и не пишет на диск, читаемый другим процессом, т.е. конфликтов нет:



Итак, когда процесс **LGWR** записывает группу 1, процесс **ARCH** читает группу 2 и записывает ее на диски архива. Когда процесс **LGWR** записывает группу 2, процесс **ARCH** читает группу 1 и записывает на диски архива. Таким образом, каждый из процессов **LGWR** и **ARCH** использует собственные отдельные устройства и не конфликтует с другими процессами.

Файлы журнала повторного выполнения — лишь один из наборов файлов Oracle, эффективность использования которых повышается при размещении на неформатированных устройствах. Если на таких устройствах можно разместить только один набор файлов, это должны быть файлы журнала повторного выполнения. Постоянно идут активные дискуссии о преимуществах и недостатках использования неформатированных устройств по сравнению с файлами в файловых системах. Поскольку эта книга не посвящена задачам администратора базы данных/системного администратора, мы не будем в них вступать. Я просто уточню, что, если предполагается использование неформатированных устройств, лучше всего на них разместить именно файлы журнала повторного выполнения. Выполнять резервное копирование активных файлов журнала повторного выполнения никогда не придется, поэтому факт их размещения на неформатированных разделах вместо файловой системы никак не скажется на сценариях резервного копирования. Процесс **ARCH** всегда будет преобразовывать "неформатированные" журналы в обычные файлы файловой системы (архивировать на неформатированные устройства нельзя), так что "сложности" работы с неформатированными устройствами в данном случае минимальны.

Временные таблицы и данные повторного выполнения/отката

Временные таблицы — новая возможность версии Oracle 8.1.5. Поэтому вокруг них много непонимания, особенно в области журнализации. В главе 6, посвященной типам таблиц базы данных, мы рассмотрим, как и зачем использовать временные таблицы. В

этом разделе мы рассмотрим только один вопрос: как работают временные таблицы с точки зрения журнализации?

Для блоков временных таблиц данные повторного выполнения не генерируются. Поэтому действия с временной таблицей "невосстановимы". При изменении блока во временной таблице запись этих изменений в файлы журнала повторного выполнения не выполняется. Однако при работе с временными таблицами генерируется данные отмены, и эти изменения в сегменте отката журнализируются. Поэтому при работе с временными таблицами генерируется некоторый объем данных повторного выполнения. На первый взгляд это кажется абсолютно бессмысленным: зачем генерировать данные отката? Потому, что в транзакции можно откатиться до точки сохранения. Можно удалить последних 50 вставленных во временную таблицу строк, но не первых 50. Для временных таблиц можно задавать ограничения и все прочие конструкции, доступные для обычных таблиц. Может произойти сбой на 500 строке оператора вставки 500 строк, т.е. потребуется откат оператора. Поскольку временные таблицы ведут себя так же, как "обычные", при работе с ними обязательно должны генерироваться данные отката. Поскольку данные в сегменте отката должны журнализироваться, при генерировании данных отката будет генерироваться и некоторый объем данных повторного выполнения.

Все не так страшно, как кажется. С временными таблицами используются в основном SQL-операторы INSERT и SELECT. К счастью, операторы INSERT генерируют немного данных отмены (блок необходимо восстановить в состояние "ничего", а для хранения "ничего" надо не так уж много места), а операторы SELECT данные отмены вообще не генерируют. Поэтому, если временные таблицы используются исключительно для вставки и запросов, этот раздел вполне можно проигнорировать. Учитывать генерирование данных отмены надо только при использовании операторов UPDATE или DELETE.

Я создал небольшой тестовый пример, демонстрирующий объем генерируемых при работе с временными таблицами данных повторного выполнения и показывающий косвенно, сколько генерируется данных отмены, поскольку для них регистрируются только изменения в сегменте отката. Для этого используются "постоянная" и "временная" таблицы идентичной структуры, с которыми выполняются одинаковые действия, а затем определяется объем сгенерированных данных повторного выполнения. Я использовал следующие простые таблицы:

```
tkyte@TKYTE816> create table perm
  2   (x char(2000) default 'x',
  3    y char(2000) default 'y',
  4    z char(2000) default 'z')
  5   /

Table created.

tkyte@TKYTE816>
tkyte@TKYTE816>
tkyte@TKYTE816> create global temporary table temp
  2   (x char(2000) default 'x',
  3    y char(2000) default 'y'.
```

```
  4    z char(2000) default 'z')
  5  on commit preserve rows
  6  /

Table created.
```

Затем, я создал небольшую хранимую процедуру для применения указанного SQL-оператора к таблице и выдачи отчета о результатах:

```
tkyte@TKYTE816> create or replace procedure do_sql(p_sql in varchar2)
  2  as
  3      l_start_redo     number;
  4      l_redo               number;
  5  begin
  6  select value into l_start_redo from redo_size;
  7
  8      execute immediate p sql;
  9      commit;
 10
 11      select value-l_start_redo into l_redo from redo_size;
 12
 13      dbms_output.put_line
 14      (to_char(l_redo, '9,999,999')  ||' bytes of redo generated for '" ||
 15      substr(replace(p_sql, chr(10), '  ') , 1, 25) || '"...');
 16  end;
 17  /

Procedure created.
```

Затем я применил к таблицам одинаковые операторы INSERT, UPDATE и DELETE:

```
tkyte@TKYTE816> set serveroutput on format wrapped
tkyte@TKYTE816> begin
  2       do_sql (' insert into perm
  3               select 1,1,1
  4                 from all_objects
  5               where rownum <= 500');
  6
  7       do_sql('insert into temp
  8               select 1,1,1
  9                 from all_objects
 10                where rownum <= 500');
 11
 12       do_sql( 'update perm set x = 2');
 13       do_sql( 'update temp set x = 2') ;
 14
 15       do_sql ('delete from perm');
 16       do_sql ('delete from temp'),;
 17  end;
 18  /
3,238,688  bytes  of redo  generated   for "insert into perm"...
   72,572 bytes  of redo  generated   for "insert into temp"...
2,166,376  bytes  of redo  generated   for "update perm set x = 2"...
```

```
1,090,336  bytes  of redo  generated   for "update temp set x = 2"...
3,320,244  bytes  of redo  generated   for "delete from perm"...
3,198,236  bytes  of redo  generated   for "delete from temp"...

PL/SQL procedure successfully completed.
```

Этот пример демонстрирует следующее.

- В процессе вставки в "реальную" таблицу генерируется много данных повторного выполнения. Для временной таблицы данные повторного выполнения практически не генерируются. В этом есть смысл: операторы вставки генерируют очень мало данных отмены и только изменения данных отмены для временных таблиц регистрируются в журнале.
- При изменении реальной таблицы генерируется примерно вдвое больше данных повторного выполнения, чем при изменении временной таблицы. И это тоже понятно. Необходимо сохранить примерно половину изменения, "предварительный образ". "Окончательный образ" (данные повторного выполнения) для временной таблицы сохранять не надо.
- При удалении строк генерируется примерно одинаковый объем данных повторного выполнения. Это понятно, потому что данные отмены для оператора DELETE имеют большой объем, а данные повторного выполнения для измененных блоков невелики. Поэтому оператор DELETE с временной таблицей работает почти так же, как и с обычной.

Поэтому при работе с временными таблицами можно использовать следующие простые правила:

- оператор INSERT генерирует мало или вообще не генерирует данные отмены/повторного выполнения;
- оператор DELETE генерирует тот же объем данных повторного выполнения, что и для обычной таблицы;
- оператор UPDATE генерирует примерно вдвое меньше данных повторного выполнения, чем для обычной таблицы.

Из этого правила есть исключения. Например, при замене пустого (NULL) столбца 2000 байт данных оператор UPDATE сгенерирует очень мало данных для отката. Такое изменение подобно вставке. С другой стороны, если столбец с 2000 байт данных становится пустым (NULL), такое изменение подобно удалению с точки зрения объема генерируемых данных повторного выполнения. В среднем можно ожидать, что при изменении временной таблицы генерируется примерно 50 процентов данных отмены/повторного выполнения от объема для обычной таблицы.

При определении объема создаваемых данных повторного выполнения можно руководствоваться здравым смыслом. Если выполняемая операция приводит к созданию данных отмены, подумайте, насколько легко или сложно будет восстановить (отменить) результат операции. Вставку 2000 байт отменить просто. Надо вернуть ситуацию, когда было 0 байт. Если удалено 2000 байт, для восстановления придется 2000 байт вставить. В этом случае генерируется значительный объем данных повторного выполнения.

Учитывая все это, избегайте удаления строк из временных таблиц. Можно использовать оператор TRUNCATE или дождаться автоматической очистки при фиксировании транзакции или удалении сеанса. Все эти методы не генерируют данные отмены и, как следствие, данные повторного выполнения. Старайтесь не изменять временные таблицы, если для этого нет серьезной причины. Временные таблицы используются в основном для вставки данных и запросов. При этом оптимально используется их уникальная возможность — не генерировать данные повторного выполнения.

Анализ данных повторного выполнения

Завершая рассмотрение файлов журнала повторного выполнения, попробуем ответить на часто задаваемый вопрос: как проанализировать эти файлы? Ранее, до появления версии Oracle 8i, это было очень сложно. Приходилось обращаться в службу поддержки Oracle, чтобы получить магическую команду для перевода содержимого файла журнала повторного выполнения в текстовый вид. В результате создавался текстовый отчет, который можно было читать и, имея некоторое представление о внутренних, недокументированных особенностях СУБД Oracle, даже приходить к каким-то выводам. С практической точки зрения это было неприемлемо, особенно если надо было проанализировать файлы журнала повторного выполнения и найти, скажем, время удаления таблицы или установки отрицательного значения столбца **SALARY** в транзакции и т.п.

И тут появился Log Miner. Речь идет о новом пакете, **DBMS_LOGMNR,** поставляемом в составе Oracle 8i и позволяющем загружать содержимое файлов журнала повторного выполнения в таблицу базы данных V$, а затем выполнять к ней SQL-запросы. Можно получить предварительное и окончательное значение каждого столбца, затронутого оператором ЯМД. Можно получить SQL-оператор, фактически "повторно выполняющий" или "отменяющий" транзакцию. В этой таблице можно найти оператор удаления из таблицы **OBJ$** (принадлежащая пользователю SYS таблица словаря данных) и увидеть, когда именно была случайно удалена таблица. Вдруг администратор базы данных сможет вернуть базу данных в состояние на момент времени, непосредственно предшествующий выполнению этого оператора **DROP TABLE** и восстановить таблицу.

Пока я просто упомяну Log Miner. В приложении А пакет **DBMS_LOGMNR** будет описан детально. Подробнее о встроенных пакетах см. в последней части книги.

Откат

Были уже затронуты многие темы, связанные с сегментами отката. Мы рассматривали, как они используются в ходе восстановления, как взаимодействуют с журналами повторного выполнения, а также упоминали, что они используются для обеспечения согласованного, не блокирующего чтения данных. В этом разделе я хочу описать проблемы, связанные с сегментами отката. Основную часть времени мы посвятим печально известной ошибке **ORA-01555: snapshot too old,** но до этого разберемся еще с двумя проблемами, касающимися отката. Сначала мы выясним, какие действия генерируют наибольший/наименьший объем данных отмены (возможно, вы и сами, после изучения предыдущих примеров с временными таблицами, сможете ответить). Затем рассмотрим,

как оператор SET **TRANSACTION** используется для указания сегмента отката, и обсудим, и каких случаях нужно его использовать. И только после этого подробно рассмотрим загадочную ошибку **ORA-01555.**

Что генерирует основной/наименьший объем данных отмены?

Ответить на этот вопрос очень легко. Наименьший объем информации отмены генерирует оператор **INSERT,** поскольку серверу Oracle достаточно записать идентификатор строки для удаления. Оператор UPDATE по этому показателю обычно идет вторым. Достаточно записать только измененные байты. Чаще всего изменяется лишь небольшая часть данных строки. Поэтому только небольшую часть строки придется записывать в сегмент отката. Многие из представленных ранее примеров демонстрировали как раз обратное, но лишь потому, что изменялись большие строки фиксированной длины, причем изменялись целиком. Гораздо чаще изменяется лишь небольшая часть строки. Оператор DELETE, как правило, генерирует наибольший объем данных отмены. Для оператора DELETE сервер Oracle должен записать в сегмент отката предварительный образ всей строки в целом. Представленный ранее пример, в котором генерировались данные повторного выполнения для временной таблицы, прекрасно это продемонстрировал. Оператор **INSERT** сгенерировал очень немного данных отмены, которую пришлось регистрировать в журнале. Оператор UPDATE сгенерировал данные, равные по объему предварительным образам измененных данных, а оператор DELETE сгенерировал всю строку данных, которая и была записана в сегмент отката.

Оператор SET TRANSACTION

SQL-оператор SET **TRANSACTION** позволяет задать сегмент отката, который должна использовать транзакция. Обычно это делается для выделения большого сегмента отката для операции, затрагивающей большой объем данных. Мне не нравится эта практика, особенно, когда ею злоупотребляют. Для нечастых множественных изменений это вполне приемлемо как разовое действие. Я, однако, считаю, что надо использовать сегменты отката одинакового размера, а выбор сегмента для транзакции предоставить системе. При использовании сегментов отката, которые могут увеличиваться (если параметр MAXEXTENTS имеет достаточно большое значение) и для которых установлен размер **OPTIMAL** (т.е. они уменьшаются до некоторого размера после существенного увеличения), отдельный большой сегмент не нужен.

Однако, если применяется один большой сегмент отката, ничто не мешает другой транзакции его использовать. Практически ничего нельзя сделать, чтобы отдать сегмент отката одной транзакции. В выделенном пространстве будут работать и другие транзакции. Гораздо проще дать системе самой выбрать сегмент отката и разрешить увеличивать его при необходимости. Кроме того, некоторые инструментальные средства не разрешают выбирать сегмент отката, например утилита **IMP** (импорт). При обновлении моментального снимка, однако, указывать сегмент отката можно, а при использовании

утилиты SQLLDR — нет. Я считаю, что все сегменты отката должны иметь достаточный размер для выполняемых в системе транзакций.

С учетом всего этого, для рассматриваемого оператора остается нечастое, однократное применение. При выполнении множественных изменений большого объема информации его использование оправдано. Создается временный сегмент отката на одном из дополнительных дисков, и выполняется изменение. После изменения этот сегмент отката отключается и удаляется. Вероятно, еще лучше сначала фрагментировать большую таблицу и выполнять изменение параллельно. В этом случае, каждый из подчиненных процессов параллельного запроса будет привязан к отдельному сегменту отката, в результате чего транзакция может использовать все сегменты отката одновременно. Этого нельзя достичь при выполнении изменения последовательно.

'ORA-01555: snapshot too old'

Ошибка ORA-01555 — одна из ошибок, сбивающих людей с толку. С ней связано много мифов, неточностей и предположений. На самом деле эта ошибка очевидна и возникает только по двум причинам, но поскольку специальный случай одной из этих причин встречается слишком часто, можно выделить три причины:

- сегменты отката слишком малы для выполняемых в системе действий;
- программы извлекают данные между фиксациями (по сути это разновидность первой причины);
- очистка блоков.

Первые две причины непосредственно связаны с моделью согласованности по чтению Oracle. Если вспомнить главу 2, посвященную архитектуре, результаты запроса "предопределены" — они хорошо известны еще до извлечения сервером Oracle первой строки. Сервер Oracle обеспечивает этот согласованный по времени моментальный снимок базы данных, используя сегменты отката для отмены изменений в блоках, изменившихся с начала выполнения запроса. Любой выполняемый оператор, например:

```
update t set x = 5 where x = 2;
insert into t select * from t where x = 2;
delete from t where x = 2;
select * from t where x = 2;
```

получит согласованное по чтению представление таблицы T и набор строк, в которых x=2, независимо от других действий, параллельно выполняемых в базе данных. Все операторы, читающие таблицу, используют этот механизм согласованности по чтению. В представленном выше примере оператор UPDATE читает таблицу, чтобы найти в ней строки, где x=2 (и затем изменить их). Оператор INSERT читает таблицу в поисках строк, где x=2, и затем вставляет их, и так далее. Использование сегментов отката для этих двух целей — отката неудачных транзакций и обеспечения согласованности по чтению — и приводит к возникновению ошибки ORA-01555.

Третья причина возникновения ошибки ORA-01555 более загадочна, поскольку в этом случае ошибка может возникать в базе данных с единственным сеансом, причем — не из-

меняющим таблицу! Это кажется невозможным; зачем нужны данные в сегменте отката для таблицы, которая не изменяется? Далее мы разберемся с этим.

Прежде чем проиллюстрировать все три причины, хочу поделиться своим опытом устранения проблем, связанных с ошибкой. ORA-1555. В общем случае возможны следующие решения.

- Проанализировать используемые объекты. Это поможет избежать третьей причины. Поскольку очистка блоков выполняется в результате множественных изменений или вставок очень большого объема данных, это все равно надо делать.
- Увеличить или добавить сегменты отката. Это уменьшает вероятность перезаписи данных отмены при выполнении продолжительного запроса. Это решение помогает устранить все три перечисленные ранее причины.
- Сократить время выполнения запроса (настроить запрос). Это наилучший выход, поэтому именно к нему следует прибегнуть в первую очередь. В результате сократится потребность в больших сегментах отката. Это решение помогает устранить все три причины возникновения ошибки.

Мы вернемся к этим решениям позже, однако в общих чертах опишем их здесь.

Сегменты отката очень маленькие

Сценарий следующий: имеется система, транзакции в которой — маленькие. Естественно, в сегментах отката требуется очень мало места. Рассмотрим, например, следующую ситуацию.

- Каждая транзакция генерирует в среднем 1 Кбайт данных отмены.
- В среднем выполняется пять таких транзакций в секунду (генерируется 5 Кбайт данных отмены в секунду, 300 Кбайт в минуту).
- В среднем раз в минуту выполняется транзакция, генерирующая 1 Мбайт данных отмены. В результате генерируется примерно 1,3 Мбайт данных отмены в минуту.
- В системе сконфигурированы сегменты отката общим размером 5 Мбайт.

Для обработки транзакции места в сегментах отката в этой базе данных предостаточно. Система будет переходить на следующий сегмент отката и использовать место в нем повторно в среднем раз в три-четыре минуты. Если размер сегментов отката определен на основе этой информации о предполагаемых транзакциях, — все в порядке.

В этой же среде, однако, надо еще создавать отчеты. Некоторые из запросов выполняются весьма долго, — скажем, пять минут. Вот тут и возникает проблема. Если запросы выполняются пять минут и им понадобится представление данных на момент начала запроса, велика вероятность возникновения ошибки ORA-01555. Поскольку сегменты отката в ходе выполнения этого запроса переписываются, понятно, что часть данных отмены, сгенерированных с начала выполнения запроса, потеряна, — они были перезаписаны. Если мы столкнемся с блоком, измененным сразу после начала запроса, данные отмены для этого блока не будет найдены, и мы получим сообщение об ошибке ORA-01555.

Рассмотрим небольшой пример. Пусть имеется таблица с блоками 1, 2, 3,… 1000000. Вот последовательный список событий, которые могли произойти:

Время (минут: секунд)	*Действие*
0:00	Начато выполнение запроса.
0:01	Другой сеанс изменяет блок 1000000. Данные отмены для этого изменения записываются в определенный сегмент отката.
0:01	Этот сеанс фиксирует транзакцию. Сгенерированные данные отмены все еще доступны, но могут быть перезаписаны при нехватке места в сегменте отката.
1:00	Запрос продолжает выполняться. Сейчас он просматривает блок 200000.
1:01	Выполняется много действий. В результате, сгенерировано уже более 1,3 Мбайт данных отмены.
3:00	Запрос все еще выполняется. Он уже дошел до блока 600000.
4:00	Сегменты отката начинают перезаписываться, и пространство, использовавшееся в начале выполнения запроса, в момент 0:00, начинает использоваться повторно. В частности, мы только что повторно использовали в сегменте отката место, где были записаны данные отмены для блока 1000000, измененного в момент времени 0:01.
5:00	Запрос добрался, наконец, до блока 1000000. Оказывается, блок был изменен после начала выполнения запроса. Сервер обращается к сегменту отката и пытается найти данные отмены для этого блока, чтобы получить его согласованное по времени представление. В этот момент выясняется, что необходимой информации больше нет. Возникает ошибка **ORA-01555,** и запрос заканчивается неудачно.
4:00	Сегменты отката начинают перезаписываться, и пространство, использовавшееся в начале выполнения запроса, в момент 0:00, начинает использоваться повторно. В частности, мы только что повторно использовали в сегменте отката место, где были записаны данные отмены для блока 1000000, измененного в момент времени 0:01.
5:00	Запрос добрался, наконец, до блока 1000000. Оказывается, блок был изменен после начала выполнения запроса. Сервер обращается к сегменту отката и пытается найти данные отмены для этого блока, чтобы получить его согласованное по времени представление. В этот момент выясняется, что необходимой информации больше нет. Возникает ошибка **ORA-01555,** и запрос заканчивается неудачно.

Вот и все. Если сегменты отката имеют такой размер, что могут быть перезаписаны по ходу выполнения запросов, и запросы эти обращаются к данным, которые могли быть

только при выполнении операторов **INSERT, UPDATE** и **DELETE.** Факт выполнения продолжительного запроса не заставляет сервер Oracle увеличить сегмент отката, чтобы при необходимости сохранить его данные. Это может сделать только продолжительная транзакция, изменяющая данные. В представленном примере, хотя сегменты отката могли расти, они не росли. Для этой системы надо изначально создать большие сегменты отката. Необходимо выделить постоянное пространство для сегментов отката, а не просто дать возможность расти при необходимости.

Для решения этой проблемы надо либо создать сегменты отката такого размера, чтобы они переписывались не чаще, чем один раз в шесть-десять минут, либо сделать так, чтобы запросы выполнялись не более двух-трех минут. Первое решение основано на том факте, что имеются запросы, выполняющиеся пять минут. В этом случае администратор базы данных должен увеличить объем постоянно выделяемых сегментов отката в два-три раза. Второе решение тоже вполне допустимо. Если можно ускорить выполнение запросов, это надо делать. Если блоки в сегментах отката, сгенерированные с начала запроса, никогда не перезаписываются, ошибка **ORA-01555** не возникнет.

Важно помнить, что вероятность возникновения ошибки **ORA-01555** определяется размером **наименьшего** сегмента отката в системе. Добавление одного "большого" сегмента отката не устранит проблему. Достаточно при выполнении запроса повторно использовать самый маленький сегмент отката, и сразу появляется возможность возникновения ошибки **ORA-01555.** Вот почему мне очень нравится идея создавать сегменты отката одинакового размера. При этом каждый сегмент отката — одновременно наибольший и наименьший. Вот почему я также избегаю использования сегментов отката "оптимального" размера. При сжатии вынужденно увеличенного сегмента отката выбрасывается множество данных отмены, которые могут еще понадобиться. Хотя удаляются самые старые данные в сегменте отката, определенный риск все же остается. Я предпочитаю уменьшать сегменты отката в периоды минимальной нагрузки вручную, если это вообще делается. Пожалуй, я слишком глубоко вошел в роль администратора базы данных, так что давайте перейдем к следующему случаю: ошибка **ORA-01555** вызвана тем, что размер сегментов отката для текущего объема изменений определен неправильно. Единственное решение — правильно задавать размер сегментов отката в соответствии с объемом изменений. Это не ваша ошибка, но это ваша проблема, раз уж вы с ней столкнулись. Эта ситуация аналогична нехватке временного пространства в ходе выполнения запроса. Надо либо предусмотреть достаточное количество временного пространства в системе, либо переписать запросы так, чтобы при выполнении для них требовалось меньше временного пространства.

Чтобы продемонстрировать этот эффект, можно создать небольшой тестовый пример. Ниже я создам очень маленький сегмент отката. Будет работать только один сеанс, использующий только этот сегмент отката, что практически гарантирует перезапись и многократное повторное использование выделенного пространства. Сеанс, использующий этот сегмент отката, будет изменять таблицу T. Он будет выполнять полный просмотр таблицы T, и читать ее от начала до конца. В другом сеансе мы выполним запрос к таблице T, который будет читать ее по индексу. Т.е. он будет читать строки таблицы в относительно случайном порядке. Он прочитает строку 1, затем — строку 1000, стоку

500, строку 20001 и т.д. При выполнении запроса блоки считываются в случайном порядке и, вероятно, по несколько раз. Вероятность возникновения ошибки **ORA-01555** в этом случае составляет почти 100 процентов. Итак, в одном сеансе выполним:

```
tkyte@TKYTE816> create rollback segment rbs_small
  2   storage
  3   (initial 8k next 8k
  4    minextents 2 maxextents 3)
  5   tablespace rbs_test
  6   /

Rollback segment created.

tkyte@TKYTE816> alter rollback segment rbs_small online;

Rollback segment altered.

tkyte@TKYTE816> create table t
  2   as
  3   select *
  4     from all objects
  5   /

Table created.

tkyte@TKYTE816> create index t_idx on t(object_id)
  2   /
Index created.

tkyte@TKYTE816> begin
  2             for x in (select rowid rid from t)
  3             loop
  4                       commit;
  5                       set transaction use rollback segment rbs_small;
  6                       update t
  7                           set object_name = lower(object_name)
  8                        where rowid = x.rid;
  9             end loop;
 10             commit;
 11   end;
 12   /
```

Пока выполняется этот блок PL/SQL, в другом сеансе выполним:

```
tkyte@TKYTE816> select object_name from t where object_id > 0 order by
object_id;

OBJECT_NAME

i_obj#
tab$

/91196853_activationactivation
```

```
/91196853_activationactivation
ERROR:
ORA-01555: snapshot too old: rollback segment number 10 with name
"RBS_SMALL" too small

3150 rows selected.

tkyte@TKYTE816> select count(*) from t;

  COUNT(*)

     21773
```

Как видите, после чтения около трех тысяч строк (примерно одной седьмой части данных) в произвольном порядке, мы получили-таки сообщение об ошибке ORA-01555. В этом случае ошибка была вызвана исключительно чтением таблицы T по индексу, в результате чего в случайном порядке читались блоки по всей таблице. Если бы выполнялся полный просмотр таблицы, то вероятность того, что ошибка ORA-01555 не возникла бы, гораздо больше. (Попробуйте изменить запрос SELECT на SELECT /*+ FULL(T) */ ... и посмотрите, что произойдет; в моей системе даже многократное выполнение этого запроса не приводит к возникновению ошибки ORA-1555). Поскольку операторы SELECT и UPDATE будут последовательно просматривать таблицу T, то оператор SELECT скорее всего опередит в просмотре UPDATE (оператор SELECT только читает данные, а UPDATE должен прочитать их и изменить, поэтому выполняется этот оператор медленнее). Выполняя случайные чтения, мы увеличиваем вероятность того, что оператору SELECT понадобится прочитать блок, давно измененный и зафиксированный оператором UPDATE. Этот пример демонстрирует некоторую неоднозначность ошибки ORA-01555. Ее возникновение зависит от того, как одновременно работающие сеансы читают и изменяют таблицы.

Данные извлекаются в нескольких транзакциях

Это разновидность той же ситуации. Вариант этот не отличается от предыдущего, но сеанс создает проблемы сам себе. Никаких действий со стороны других сеансов не нужно. Мы уже изучали эту ситуацию в главе 4, посвященной транзакциям, и кратко рассмотрим ее снова. Смысл в том, что извлечение строк, перемежающееся выполнением операторов COMMIT, — верный способ получить сообщение об ошибке ORA-01555. По моим наблюдениям, большинство ошибок ORA-01555 связано с такими действиями. Удивительно, но разработчики иногда реагируют на получение этой ошибки тем, что фиксируют транзакции еще чаще, поскольку в сообщении сказано, что сегмент отката — слишком мал. Предполагают, что это поможет решить проблему (исходя из ошибочного вывода, что при изменении используется слишком много места в сегменте отката), тогда как на самом деле это приводит лишь к скорейшему возникновению этой ошибки.

Итак, надо изменить большой объем информации. По некоторым причинам вы не обеспечили достаточно места в сегментах отката. Поэтому принимается решение фик-

сировать транзакцию каждые X строк, чтобы "сэкономить" место в сегментах отката. Мало того, что это медленнее, и в конечном итоге будет сгенерировано больше данных отмены и повторного выполнения, это еше и гарантированный способ получить сообщение об ошибке ORA-01555. Продолжая предыдущий пример, я могу легко это продемонстрировать. Используя ту же таблицу T, что и в примере выше, просто выполните:

```
tkyte@TKYTE816> declare
  2            l_cnt number default 0;
  3  begin
  4           for x in  (select rowid rid,  t.* from t where object_id >
0)
  5            loop
  6                    if  (mod(l_cnt,100)  = 0)
  7                     then
  8                           commit;
  9                          set transaction use rollback segment rbs_small;
 10                    end if;
 11                    update t
 12                       set object_name = lower(object_name)
 13                     where rowid = x.rid;
 14                   l_cnt  := l_cnt + 1;
 15           end loop;
 16       commit;
 17  end;
 18  /
declare
*
ERROR at line 1:
ORA-01555: snapshot too old: rollback segment number 10 with name
 "RBS_SMALL" too small
ORA-06512: at line 4
```

Здесь выполняется случайное чтение таблицы T по индексу. Мы изменяем по одной строке за раз в цикле. После изменения 100 строк транзакция фиксируется. Рано или поздно мы повторно обратимся в **запросе** к блоку, уже измененному оператором **UPDATE,** и этот блок уже не удастся восстановить по сегментам отката (поскольку данные в них давно перезаписаны). Теперь мы попали в неприятную ситуацию: процесс изменения завершился неудачей на полпути.

Можно, как было показано в главе 4, придумать более изощренный метод изменения данных. Например, найти минимальный и максимальный идентификатор **OBJECT_ID.** Можно поделить этот диапазон на поддиапазоны по 100 строк и делать изменения, записывая в другую таблицу информацию об успешно выполненных действиях. Это позволит реализовать процесс, который должен быть реализован одним оператором и одной транзакцией, в виде множества отдельных транзакций, организованных с помощью сложного процедурного кода таким образом, чтобы они могли быть перезапущены в случае сбоя. Например:

```
tkyte@TKYTE816> create table done(object_id int);

Table created.

tkyte@TKYTE816> insert into done values  (0) ;

1 row created.

tkyte@TKYTE816> declare
  2             l_cnt number;
  3             l_max number;
  4   begin
  5             select object_id into l_cnt from done;
  6             select max(object_id)  into l_max from t;
  7
  8             while  (l_cnt < l_max)
  9             loop
 10                       update t
 11                          set object_name = lower(object_name)
 12                        where object_id > l_cnt
 13                          and object_id <= l_cnt+100;
 14
 15                       update done set object_id = object_id+100;
 16
 17                       commit;
 18                       set transaction use rollback segment rbs_small;
 19                      l_cnt  := l_cnt + 100;
 20             end loop;
 21   end;
 22   /

PL/SQL procedure successfully completed.
```

Мы предложили сложное решение, которое надо еще тестировать и анализировать. И работать оно будет намного медленнее, чем простой оператор:

```
update t set object_name = lower(object_name)  where object_id > 0;
```

Простое и, по моему мнению, правильное решение этой дилеммы, — сконфигурировать для системы сегменты отката достаточного размера и использовать один оператор UPDATE. При изменении большего, чем обычно, объема данных используйте "черновой" сегмент отката, создаваемый специально для больших процессов, и удалите его после этого. Это решение гораздо лучше сложного процедурного решения, которое может не сработать просто в силу своей сложности (из-за ошибки программирования). Проще дать системе сделать все необходимое, чем пытаться придумать сложные обходные пути, позволяющие "сэкономить место".

Отложенная очистка блоков

Эту причину ошибки ORA-01555 полностью устранить сложнее, но она встречается реже, поскольку слишком редки обстоятельства, при которых она возникает (по крайней мере, в версии Oracle 8i). Мы уже рассматривали механизм очистки блоков: сеанс,

обратившийся к измененному блоку, проверяет, активна ли еще транзакция, изменившая его. Если оказалось, что транзакция не активна, он очищает блок так, что следующему обращающемуся к нему сеансу не придется выполнять этот процесс. Чтобы очистить блок, сервер Oracle находит (по заголовку блока) сегмент отката, использованный в предыдущей транзакции, а затем проверяет по заголовку сегмента отката, была ли транзакция зафиксирована. Эта проверка выполняется одним из двух способов. В одном из них используется способность сервера Oracle определить, что транзакция зафиксирована, даже если ее слот в таблице транзакций сегмента отката был перезаписан. Другой способ — найти в таблице транзакций сегмента отката номер зафиксированного изменения **(COMMIT** SCN), что свидетельствует о недавней фиксации транзакции, поскольку ее слот еще не перезаписан.

При отложенной очистке блока ошибка **ORA-01555** может возникать в следующих случаях.

- Изменение выполнено и зафиксировано, а блоки не очищены автоматически (например, транзакция изменила больше блоков, чем поместится в 10 процентах пространства буферного кэша в области SGA).
- Эти блоки не затронуты другим сеансом: их затронет наш невезучий запрос.
- Начался "продолжительный" запрос. Он рано или поздно прочитает некоторые из описанных выше блоков. Этот запрос начинается при значении **SCN** t1. Именно до этого номера изменения в системе придется откатывать данные для того, чтобы обеспечить согласованность чтения. Запись для транзакции, изменившей данные, в момент начала выполнения запроса еще находится в таблице транзакций сегмента отката.
- В ходе выполнения запроса в системе зафиксировано много транзакций. Эти транзакции не затронули рассматриваемые блоки (если бы затронули, мы бы с этой проблемой не столкнулись).
- Таблица транзакций в сегменте отката переполняется, и ее слоты используются повторно из-за большого количества зафиксированных транзакций. Что хуже всего, запись исходно изменившей данные транзакции тоже была использована повторно и переписана. Кроме того, система повторно использовала экстенты в сегменте отката, что не позволяет согласовать по чтению блок заголовка сегмента отката.
- Наименьший номер изменения **SCN,** записанный в сегменте отката, теперь превосходит tl (он больше, чем согласованный по чтению номер **SCN** для запроса) из-за большого количества зафиксированных транзакций.

Теперь, когда наш запрос доберется до блока, измененного и зафиксированного до его начала, возникнут проблемы. Обычно запрос обращается к сегменту отката, на который указывает блок, и узнает состояние изменившей его транзакции (другими словами, находит номер изменения **SCN** для оператора фиксации транзакции). Если это значение SCN меньше, чем t1, запрос может использовать этот блок. Если же значение номера изменения **SCN** для оператора **COMMIT** превосходит t1, запрос должен отка-

тить этот блок. Проблема, однако, в том, что запрос в этом случае не может определить: больше номер изменения COMMIT SCN для блока, чем t1, или меньше. Непонятно, можно использовать блок или нет. В результате выдается сообщение об ошибке ORA-01555.

Можно искусственно вызвать возникновение этой ошибки с помощью одного сеанса, но более впечатляющим получится результат при использовании двух сеансов, поскольку можно будет показать, что ошибка не вызвана извлечением данных в нескольких транзакциях. Мы продемонстрируем оба примера, поскольку они невелики по объему и очень похожи.

Создадим много блоков в таблице, которые требуют очистки. Затем откроем курсор для таблицы и в небольшом цикле инициируем множество транзакций. Я помещаю все изменения в один и тот же сегмент отката, чтобы гарантировано получить нужную ошибку. Рано или поздно произойдет ошибка ORA-01555, поскольку внешний запрос в цикле (SELECT * FROM T) столкнется с проблемой очистки блока из-за частого изменения и фиксации во внутреннем цикле:

```
tkyte@TKYTE816> create table small(x int) ;

Table created.

tkyte@TKYTE816> insert into small values  (0) ;

1 row created.

tkyte@TKYTE816> begin
  2     commit;
  3     set transaction use rollback segment rbs_small;
  4     update t
  5        set object_type = lower(object_type);
  6     commit;
  7
  8          for x in  (select * from t)
  9          loop
 10                 for i in 1 . . 20
 11                  loop
 12                          update small set x = x+1;
 13                          commit;
 14                          set transaction use rollback segment rbs_small;
 15                  end loop;
 16          end loop;
 17  end;
 18  /
begin
*
ERROR at line 1:
ORA-01555: snapshot too old: rollback segment number 10 with name
"RBS_SMALL" too small
ORA-06512: at line 8

tkyte@TKYTE816> select * from small;
```

X

196900

Для возникновения представленной выше ошибки я использовал ту же таблицу T с примерно 22000 строк, как и в прежних примерах (в моей системе она занимает около 300 блоков). У меня сконфигурирован буферный кэш размером 300 блоков (10 процентов от него составляет 30 блоков). Оператор UPDATE в строке 4 должен оставить около 270 блоков, требующих очистки. Затем мы выполняем оператор SELECT * из таблицы, в которой требуется очистить множество блоков. Для каждой извлекаемой строки выполняется 20 транзакций, причем я предусмотрел, чтобы эти транзакции направлялись в тот же сегмент отката, что и первоначальный оператор UPDATE (конечная цель — переписать его слот в таблице транзакций). После обработки около 10000 строк в запросе к таблице T (о чем свидетельствует значение в таблице SMALL) мы получили сообщение об ошибке ORA-01555.

Теперь, чтобы показать, что это не связано с извлечением данных в нескольких транзакциях (это вполне возможно, поскольку при извлечении данных выполняются операторы COMMIT), мы используем два сеанса. Первый сеанс мы начнем с создания таблицы **STOP_OTHER_SESSION**. Ее мы будем использовать для уведомления другого сеанса, генерирующего множество транзакций, что пора остановиться. Затем, сеанс изменит множество блоков в таблице еще раз, как в предыдущем примере, и начнет продолжительный запрос к таблице. Процедура **DBMS_LOCK. SLEEP** используется для реализации ожидания между извлечениями строк, что увеличивает время выполнения запроса. Это имитирует время выполнения, необходимое для сложной обработки каждой строки:

```
tkyte@TKYTE816> create table stop_other_session (x int) ;

Table created.

tkyte@TKYTE816> declare
  2            l_cnt number  := 0;
  3  begin
  4       commit;
  5       set transaction use rollback segment rbs small;
  6       update t
  7          set object_type  - lower(object_type);
  8       commit;
  9
 10       for x in  (select  *  from t)
 11       loop
 12            dbms_lock.sleep(1);
 13       end loop;
 14  end;
 15  /
```

Теперь, пока выполняется представленный выше код, в другом сеансе необходимо выполнить следующее:

```
tkyte@TKYTE816> create table small(x int) ;

Table created.

tkyte@TKYTE816> insert into small values (0) ;

1 row created.

tkyte@TKYTE816> begin
  2          commit;
  3          set transaction use rollback segment rbs_small;
  4          for i in 1 .. 500000
  5          loop
  6                  update small set x = x+1;
  7          commit;
  8          set transaction use rollback segment rbs_small;
  9          for x in (select * from stop_other_session)
 10          loop
 11            return; — остановиться, когда другой сеанс попросит это
                       — сделать
 12          end loop;
 13       end loop;
 14  end;
 15  /

PL/SQL procedure successfully completed.
```

Через некоторое время в первом сеансе будет получено такое сообщение:

```
declare
*
ERROR at line 1:
ORA-01555: snapshot too old: rollback segment number 10 with name
"RBS_SMALL" too small
ORA-06512: at line 10
```

Сообщение об ошибке — то же самое, но теперь мы не извлекали данные в нескольких транзакциях, т.е. читаемые данные никем не изменялись.

Как я уже упоминал, это — редкий случай. Необходимо стечение многих обстоятельств. Необходимо наличие блоков, требующих очистки, а такие блоки в версии Oracle8i встречаются редко (это случалось в версиях 7.x и 8.0). Оператор сбора статистики ANALYZE избавляет от этих блоков, поэтому большинство обычных причин их возникновения, в частности множественные изменения и загрузку большого объема данных, можно не учитывать, поскольку после таких действий таблицы все равно надо анализировать. Транзакции затрагивают менее 10 процентов блоков в буферном кэше, т.е. не генерируют блоки, требующие очистки. Если вы уверены, что столкнулись именно с этой проблемой (оператор SELECT к таблице, к которой не применяются при этом операторы ЯМД, заканчивается сообщением об ошибке ORA-01555), попытайтесь сделать следующее.

- Прежде всего убедитесь, что используете транзакции соответствующего размера. Проверьте, что не фиксируете транзакции чаще, чем это необходимо.

- Проанализируйте соответствующие объекты. Поскольку очистка блоков требуется в результате очень больших множественных изменений или вставок данных, это все равно надо сделать.
- Увеличьте размер сегментов отката или создайте дополнительные. Благодаря этому уменьшится вероятность перезаписи слота таблицы транзакций сегмента отката в случае продолжительного запроса. Кроме того, это позволит избежать других причин возникновения ошибки ORA-01555 (фактически эти две причины тесно связаны; при обработке запроса вы сталкиваетесь с повторным использованием сегмента отката).
- Сократите время выполнения запроса (настройте его). В случае успеха это будет удачным решением, поэтому именно с этого и стоит начать.

Резюме

В этой главе мы рассмотрели значение для разработчика данных повторного выполнения и отмены. Здесь я представил в основном те проблемы, которые касаются администратора базы данных или системного администратора. Я показал важность данных повторного выполнения и отмены и продемонстрировал, что она не является излишней тратой ресурсов: журналы повторного выполнения и сегменты отката являются неотъемлемыми и необходимыми компонентами базы данных. Глубоко понимая их работу и назначение, вы сможете использовать их с максимальной отдачей. Кроме того, вы должны усвоить, что при более частой, чем необходимо, фиксации ничего не "экономится" (фактически ресурсы транжирятся: необходимо больше процессорного времени, больше места на диске и больше программирования). Разберитесь, что должен делать сервер баз данных, и позвольте ему функционировать нормально.

6

Таблицы

В этой главе мы рассмотрим таблицы базы данных. Будут представлены различные типы таблиц и показано, в каких случаях имеет смысл использовать каждый из них. Будут детально описаны особенности физического хранения таблиц и организацию их данных.

Когда-то был только один тип таблиц — "обычные". Они управлялись как "куча" (ее определение представлено ниже). Со временем в Oracle добавились более сложные типы таблиц. Теперь, кроме обычных таблиц, есть еще кластерные (двух типов), таблицы, организованные по индексу, вложенные таблицы, временные и объектные. Каждый тип таблиц имеет свои характеристики, делающие его наиболее подходящим в тех или иных прикладных областях.

Типы таблиц

Прежде чем перейти к деталям, определим каждый тип таблиц. В Oracle 8i — семь основных типов таблиц.

- **Таблицы, организованные в виде кучи.** Это "обычные", стандартные таблицы базы данных. Данные управляются по принципу "кучи". При добавлении данных используется первое же свободное место в сегменте, достаточное для их размещения. При удалении данных из таблицы освободившееся место может повторно использоваться следующими операторами **INSERT** и **UPDATE.** Вот откуда название "организованные в виде кучи" для таких таблиц. "Куча" — это пространство памяти, используемое достаточно случайным образом.
- **Таблицы, организованные по индексу.** Такая таблица хранится в структуре индекса. Это предполагает физическое упорядочение ее строк. Если в обычной таблице данные вставляются в любое свободное место, в таблице, организованной по индексу, хранимые данные отсортированы по первичному ключу.

- **Таблицы** в **кластере.** Хранение таблицы в кластере дает два преимущества. Во-первых, несколько таблиц можно хранить вместе. Обычно в блоке хранятся данные только одной таблицы. В кластере же в одном блоке могут храниться данные нескольких таблиц. Во-вторых, все данные, содержащие одно и то же значение ключа кластера, также хранятся вместе. Данные "кластеризованы" вокруг значения ключа кластера. Ключ кластера строится с помощью индекса на основе B*-дерева.
- **Таблицы** в **хеш-кластере.** Аналогичны представленным ранее таблицам в кластере, но, вместо индекса на основе B*-дерева, для поиска блока данных по ключу кластера используется хеширование ключа. В хеш-кластере сами данные (образно говоря) и есть индекс. Это хорошо подходит для чтения данных, соответствующих определенному значению ключа.
- **Вложенные таблицы.** Часть объектно-реляционных расширений сервера Oracle. Это просто генерируемые и поддерживаемые системой дочерние таблицы связанные как предок-потомок. Они устроены аналогично таблицам **EMP** и **DEPT** в схеме **SCOTT.** Таблицу **EMP** можно считать дочерней по отношению к таблице **DEPT,** поскольку в таблице **EMP** есть внешний ключ, **DEPTNO,** ссылающийся на таблицу **DEPT.** Главное различие в том, что это — не "отдельные" таблицы, как **EMP.**
- **Временные таблицы. В** этих таблицах сохраняются черновые данные на время транзакции или сеанса. При необходимости для этих таблиц выделяются временные экстенты из временного табличного пространства пользователя. Каждый сеанс будет "видеть" только выделенные им самим экстенты и никогда не "увидит" данные, созданные в других сеансах.
- **Объектные таблицы.** Это таблицы, создаваемые на основе объектного типа. Они имеют специальные атрибуты, отсутствующие у необъектных таблиц, например генерируемый системой псевдостолбец **REF** (идентификатор объекта) для каждой строки. Объектные таблицы фактически являются отдельным случаем обычных, организованных по индексу и временных таблиц, и могут включать вложенные таблицы.

Имеется несколько общих свойств таблиц, не зависящих от их типа.

- Таблица может иметь до 1000 столбцов, хотя я не рекомендовал бы использовать такие таблицы без крайней необходимости. Таблицы наиболее эффективно работают при количестве столбцов, намного меньшем, чем 1000.
- Таблица может иметь практически неограниченное количество строк. Хотя при этом придется столкнуться с другими ограничениями. Например, табличное пространство может обычно состоять не более чем из 1022 файлов. Пусть используются файлы размером 32 Гбайт, тогда получаем 32704 Гбайт в каждом табличном пространстве. Это означает — 2143289344 блока по 16 Кбайт каждый. В один такой блок можно вместить 160 строк размером от 80 до 100 байт. Это дает в итоге 342926295040 строк. Однако если фрагментировать таблицу, это количество можно увеличить в десять раз. Теоретически ограничения, конечно, есть, но прежде чем они будут достигнуты, придется столкнуться с другими, практическими ограничениями.

- Таблица может иметь столько индексов, сколько имеется перестановок столбцов (и перестановок функций от этих столбцов), но не более 32 столбцов, хотя и в этом случае будут практические ограничения на количество реально создаваемых и сопровождаемых индексов.
- Нет ограничения на количество таблиц. И в этом случае практические ограничения будут держать количество таблиц в разумных границах. Миллионов таблиц у вас не будет (такое количество сложно создать и поддерживать), но тысячи таблиц поддерживаются элементарно.

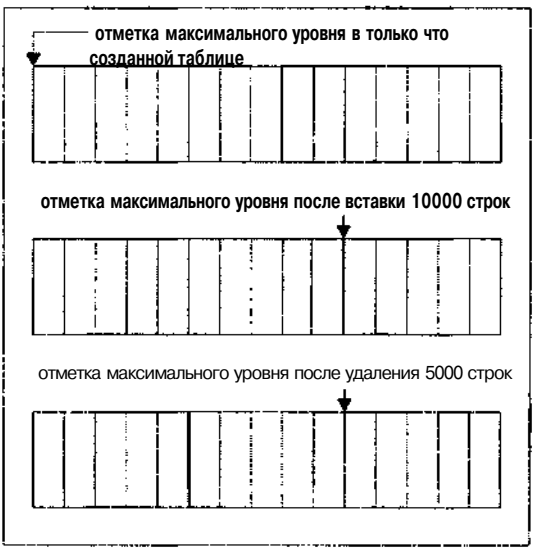
Начнем с рассмотрения параметров и терминов, относящихся к таблицам. После этого будут описаны простые таблицы, организованные в виде "кучи", а затем — и другие типы таблиц.

Терминология

В этом разделе мы рассмотрим различные параметры хранения и терминологию, связанную с таблицами. Не все описываемые параметры используются для каждого типа таблиц. Например, параметр PCTUSED не имеет смысла для таблицы, организованной по индексу. Применимые параметры будут указаны при описании каждого типа таблиц. Цель этого раздела — представить термины и дать их определения. При необходимости, более детальная информация об использовании конкретных параметров будет представлена в следующих разделах.

Отметка максимального уровня

Этот термин (high water mark) используется для объектов, хранящихся в базе данных. Если представить себе таблицу как "линейную" структуру в виде последовательности блоков слева направо, то *отметкой максимального уровня* будет крайний правый блок, когда-либо содержавший данные. Например:



Этот рисунок показывает, что отметка максимального уровня сначала указывает на первый блок только что созданной таблицы. По мере добавления данных в таблицу и использования новых блоков отметка максимального уровня повышается. Если удалить некоторые (или даже **все)** строки таблицы, может появиться много блоков, уже не содержащих данные, но они все равно находятся до отметки максимального уровня и останутся там, пока объект не будет пересоздан или усечен.

Отметка максимального уровня важна, поскольку сервер Oracle при полном просмотре будет сканировать все блоки до этой отметки, даже если они не содержат данных. Это скажется на скорости выполнения полного просмотра, особенно если большая часть блоков до отметки максимального уровня — пустые. Чтобы убедиться в этом, просто создайте таблицу из 1000000 (или другого большого количества) строк. Выполните запрос **SELECT COUNT(*)** к этой таблице. Теперь удалите все строки таблицы и убедитесь, что запрос **SELECT COUNT(*),** не возвращающий строк, работает так же долго, как и в случае возвращения 1000000 строк. Так происходит потому, что сервер Oracle просматривает все блоки до отметки максимального уровня в поисках данных. Сравните это с результатом применения оператора **TRUNCATE.** При выполнении **TRUNCATE** отметка максимального уровня будет сброшена "в ноль". Если предполагается удаление всех строк таблицы, именно по этой причине надо использовать оператор **TRUNCATE.**

Списки свободных мест

В *списке свободных мест* **(FREELIST)** сервер Oracle отслеживает блоки объекта до отметки максимального уровня, в которых есть свободное пространство. С каждым объектом будет связан хотя бы один список свободных мест, и по мере использования блоков они будут при необходимости добавляться или удаляться из этого списка. Важно отметить, что в списке свободных мест будут только блоки, находящиеся до отметки максимального уровня. Блоки за отметкой максимального уровня будут использоваться, только когда списки свободных мест пусты, — в этот момент сервер Oracle переносит отметку максимального уровня дальше и добавляет соответствующие блоки в список свободных мест. Таким образом, сервер Oracle откладывает увеличение отметки максимального уровня для объекта, пока это не станет действительно необходимым.

У объекта может быть несколько списков свободных мест. Если предполагается выполнение значительного количества операций **INSERT** или **UPDATE** для объекта множеством одновременно работающих пользователей, создание нескольких списков свободных мест может существенно повысить производительность (за счет возможного использования дополнительной памяти). Как будет показано далее, наличие достаточного количества списков свободных мест принципиально важно.

В среде с множеством одновременных вставок или изменений данных списки свободных мест могут как снижать так и повышать производительность. Предельно простой тест может показать преимущества установки подходящего количества списков свободных мест. Берем простейшую таблицу:

```
tkyte@TKYTE816> create table t (x int);
```

и с помощью двух сеансов начинаем интенсивно вставлять в нее данные. Если измерить общесистемное количество ожиданий событий, связанных с блоками, до и после

этой операции, можно выявить продолжительные ожидания доступности блоков данных (при вставке данных). Это зачастую вызвано недостаточным количеством списков свободных мест в таблицах (и в индексах, но к индексам мы еще вернемся в главе 7). Итак, я создал временную таблицу:

```
tkyte@TKYTE816> create global temporary table waitstat_before
  2  on commit preserve rows
  3  as
  4  select * from v$waitstat
  5  where 1=0
  6  /

Table created.
```

для хранения исходного количества ожиданий блоков. Затем в двух сеансах я одновременно выполнил:

```
tkyte@TKYTE816> truncate table waitstat_before;

Table truncated.

tkyte@TKYTE816> insert into waitstat_before
  2  select * from v$waitstat
  3  /

14 rows created.

tkyte@TKYTE816> begin
  2           for i in 1 .. 100000
  3            loop
  4                     insert into t values (i) ;
  5                      commit;
  6            end loop;
  7  end;
  8/

PL/SQL procedure successfully completed.
```

Это — очень простой блок кода, и мы — единственные пользователи базы данных в этот момент. Производительность при этом должна быть максимальной. У меня создан большой буферный кэш, журналы повторного выполнения имеют нужный размер, индексы не замедляют работу; все должно работать быстро. В результате, однако, я получаю следующее:

```
tkyte@TKYTE816> select a.class, b.count-a.count count, b.time-a.time time
  2    from waitstat_before a, v$waitstat b
  3   where a.class = b.class
  4  /

CLASS                COUNT        TIME

bitmap block             0           0
```

```
bitmap index block          0            0
data block               4226         3239
extant map                  0            0
free list                   0            0
save undo block             0            0
save undo header            0            0
segment header              2            0
sort block                  0            0
system undo block           0            0
system undo header          0            0
undo block                  0            0
undo header               649           36
unused                      0            0
```

В ходе одновременной работы этих сеансов пришлось ждать более 32 секунд. Это целиком следствие недостаточного количества списков свободных мест для одновременного выполнения подобного рода действий. Можно легко вообще избавиться от этого ожидания, просто создав таблицу с несколькими списками свободных мест:

```
tkyte@TKYTE816> create table t (x int) storage (FREELISTS 2);

Table created.
```

или изменив уже существующий объект:

```
tkyte@TKYTE816> alter table t storage (FREELISTS 2);

Table altered.
```

Теперь можно убедиться, что оба показанных ранее времени ожидания свелись к нулю. Для таблицы необходимо попытаться определить максимальное количество одновременных вставок или изменений, для выполнения которых потребуется выделение дополнительного пространства. Под одновременностью я имею в виду ситуации, когда несколько сеансов в один и тот же момент будут запрашивать свободный блок для соответствующей таблицы. Речь не идет о количестве одновременно выполняющихся транзакций — нас интересует, сколько сеансов будет одновременно выполнять вставку, независимо от границ транзакций. Для достижения максимального параллелизма необходимо как минимум столько списков свободных мест, сколько вставок одновременно выполняется в таблицу.

Значит, задав большое значение параметра FREELIST, можно забыть про все эти проблемы, верно? Нет, — это было бы слишком просто. Каждый процесс будет использовать один список свободных мест. Он не будет при поиске свободного места переходить от одного списка к другому. Это означает, что если в списке, используемом процессом, свободного места не осталось, он не будет искать свободное место в другом списке. И хотя в таблице имеется десять списков свободных мест, будет передвинута отметка максимального уровня для таблицы. Если же передвигать отметку некуда (все пространство использовано), будет выделен новый экстент. Затем процесс продолжит использовать пространство только в своем списке свободных мест (который сейчас пуст). При использовании нескольких списков свободных мест приходится идти на компромисс. С

одной стороны, несколько списков свободных мест позволяют существенно повысить производительность. С другой стороны, таблица будет занимать больше пространства на диске, чем необходимо. Придется решать, что важнее.

Не стоит недооценивать полезность этого параметра, особенно с учетом возможности изменять его при желании в версиях, начиная с Oracle 8.1.6. Можно увеличить его значение для параллельной загрузки данных в обычном режиме с помощью утилиты **SQLLDR.** При этом будет достигнута высокая степень параллелизма загрузки с минимальными ожиданиями. После загрузки можно вернуть прежнее, более подходящее для обычной работы, значение параметра **FREELIST.** При этом блоки из нескольких существующих списков свободных мест будут объединены в один основной список свободных мест.

Параметры PCTFREE и PCTUSED

Эти два параметра управляют добавлением и удалением блоков из списков свободных мест. При задании для таблицы (но не для таблицы, организованной по индексу, как будет показано далее) параметр **PCTFREE** сообщает серверу Oracle, сколько места должно быть зарезервировано в блоке для будущих изменений. Стандартное значение — 10 процентов. Предположим, используются блоки размером 8 Кбайт. Когда при добавлении новой строки в блок свободного места в блоке останется меньше 800 байт, сервер Oracle будет использовать новый блок вместо существующего. Эти 10 процентов пространства данных в блоке оставляются для изменений строк блока. Если строку придется изменять, в блоке будет место для размещения измененной строки.

Итак, если параметр **PCTFREE** определяет, когда сервер Oracle убирает блок из списка свободных мест, чтобы в него больше не вставлялись строки, то параметр **PCTUSED** определяет, когда сервер Oracle снова **вернет** блок в список свободных мест. Если параметр **PCTUSED** установлен равным 40 процентам (стандартное значение) и в блоке достигнут уровень заполнения **PCTFREE** (блок уже не находится в списке свободных мест), то сервер Oracle вернет его в список, только когда в блоке станет свободным 61 процент пространства. При использовании стандартных значений параметров **PCTFREE (10)** и **PCTUSED** (40) блок будет оставаться в списке свободных мест, пока не заполнится на 90 процентов (в нем останется 10 процентов свободного пространства). Как только он заполнится на 90 процентов, то будет удален из списка свободных мест и не попадет в него, пока свободное пространство не составит 60 процентов блока.

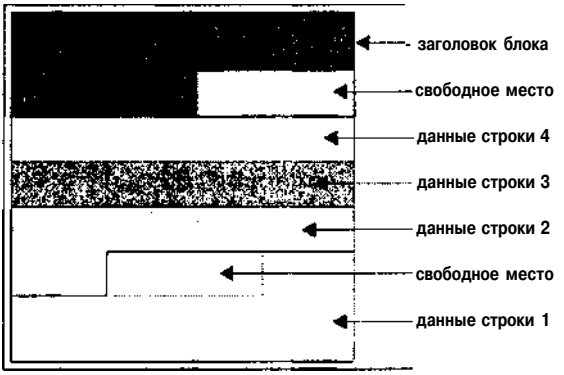
Параметры **PCTFREE** и **PCTUSED** реализуются по-разному для разных типов таблиц, на что при описании этих типов я буду обращать внимание. Для некоторых типов таблиц используются оба параметра, для других — только **PCTFREE,** да и то лишь при создании объекта.

Есть три значения параметра **PCTFREE:** слишком большое, слишком маленькое и подходящее. Если установить слишком большое значение параметра **PCTFREE** для блоков, будет напрасно расходоваться пространство на диске. Если установить значение **PCTFREE** равным 50 процентам, а данные никогда не изменяются, то 50 процентов каждого блока просто пустуют. В другой таблице, однако, значение 50 процентов может быть вполне уместным. Если первоначально маленькие строки со временем увеличиваются

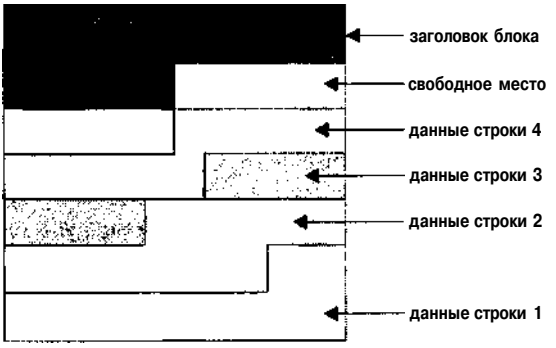
примерно вдвое, большое значение параметра PCTFREE позволит избежать переноса строк.

Перенос строки

Итак, когда переносится строка? Строка переносится из блока, в котором она была создана, когда она выросла настолько, что уже не помещается в блоке с остальными строками. Перенос строки проиллюстрирован ниже. Все начинается с блока, который выглядит примерно так:

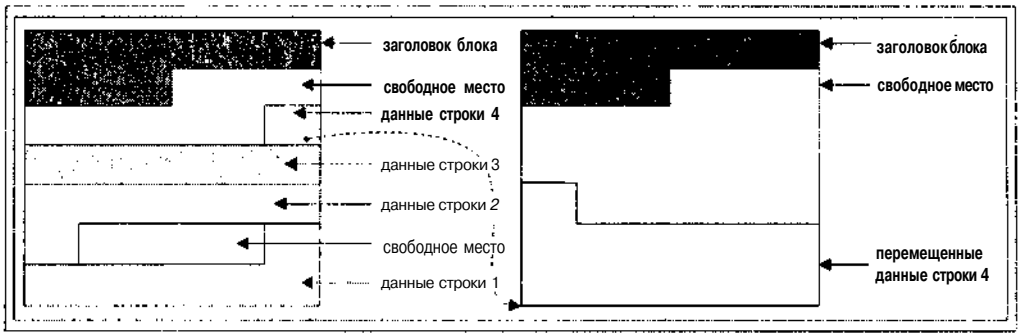


Примерно одна седьмая блока — свободное место. Однако необходимо более чем в два раза увеличить занимаемое строкой 4 место с помощью оператора UPDATE (сейчас эта строка занимает одну седьмую часть блока). В данном случае, даже если сервер Oracle объединит все свободное пространство в блоке в один фрагмент следующим образом:



все равно для увеличения строки 4 более чем в два раза места не хватит, потому что общее свободное пространство меньше, чем текущий размер строки 4. Если бы строка могла поместиться в свободное пространство после его объединения, то это объединение было бы выполнено. В нашем случае, однако, сервер Oracle не будет выполнять объединение, и блок останется неизменным. Поскольку строка 4 должна выйти за границы блока, сервер Oracle перенесет ее. Но она, однако, не будет просто перемещена; ее не-

обходимо оставить как "адрес для пересылки". Могут существовать индексы, физически ссылающиеся на текущий адрес строки 4. При простом изменении эти индексы не будут обновляться. (Учтите, что в случае использования фрагментированных таблиц идентификатор, или адрес, строки будет меняться; этот особый случай мы рассмотрим в главе 14, посвященной фрагментации.) Поэтому когда сервер Oracle переносит строку, он оставляет по старому адресу указатель на ее реальное местонахождение. После изменения блоки могут выглядеть примерно так:



Итак, вот что такое перенесенная строка: это строка, которую пришлось перенести из блока, в который она была вставлена, в какой-то другой блок. Почему это надо учитывать? Приложение никогда о переносе строки не "узнает", SQL-операторы менять не придется. Это влияет только на производительность. Если придется читать эту строку через индекс, он будет ссылаться на исходный блок. Этот блок, в свою очередь, будет ссылаться на новый блок. Вместо выполнения, скажем, двух операций ввода/вывода для чтения индекса и одной для чтения таблицы, для получения фактических данных строки потребуется еще одна дополнительная операция ввода/вывода блока. Сама по себе это "небольшая" проблема; вы эту дополнительную операцию даже не заметите. Однако, если в этом состоянии окажется значительная часть строк и обращаться к таблице будет множество пользователей, этот побочный эффект будет заметен. Доступ к данным начнет замедляться (дополнительная операция ввода/вывода увеличивает время доступа), эффективность буферного кэша — снижаться (придется буферизовать в два раза больше блоков, чем при отсутствии переноса строк), а размер и сложность таблицы — расти. Именно по этим причинам переноса строк следует избегать. Интересно разобраться, что сервер Oracle будет делать, если перенесенную с левого блока (на представленной ранее схеме) в правый блок строку **снова** придется переносить. Это может понадобиться, поскольку в блок, куда она была перенесена, добавлены строки, а наша строка опять увеличивается при изменении. Сервер Oracle перенесет строку назад в исходный блок и, если в нем достаточно места, оставит ее там (строка становится "неперенесенной"). Если места в исходном блоке недостаточно, сервер Oracle перенесет всю строку в другой блок и изменит адрес перенаправления в **исходном** блоке. Таким образом, перенос строк всегда создает один уровень косвенной ссылки. Итак, теперь мы можем вернуться к параметру **PCTFREE** и его использованию; при правильной установке значения этого параметра можно свести перенос строк к минимуму.

Установка значений PCTFREE и PCTUSED

Установка значений параметров PCTFREE и PCTUSED — важная тема, которой не всегда уделяется должное внимание. Я хочу продемонстрировать, как можно оценить дисковое пространство, используемое объектами схемы. Я буду использовать хранимую процедуру, демонстрирующую последствия вставки в таблицу данных при различных значениях параметров PCTFREE/PCTUSED, после чего выполняется ряд изменений одних и тех же данных. Это покажет, как эти параметры могут повлиять на количество блоков, доступных в списке FREELIST (что, в конечном итоге, влияет на использование пространства, количество перенесенных строк и т.д.). Предлагаемые сценарии — иллюстративные; они не определяют оптимальные значения параметров. Их можно использовать, чтобы разобраться, как сервер Oracle работает с блоками при выполнении различных изменений. Для эффективного использования эти шаблоны сценариев придется изменить.

Я начну с создания тестовой таблицы:

```
tkyte@TKYTE816> create table t (x int, y char(1000) default 'x');

Table created.
```

Она очень проста, но для демонстрации прекрасно подойдет. Использование типа CHAR гарантирует, что все строки с непустым значением в столбце Y будут иметь длину чуть более 1000 байт. Поэтому можно предположить, что будет происходить с блоками определенного размера. Теперь представлю процедуру для оценки списка FREELIST и использования пространства в блоке:

```
tkyte@TKYTE816> create or replace procedure measure_usage
  2  as
  3        l_free_blks                 number;
  4        l_total_blocks              number;
  5        l_total_bytes               number;
  6        l_unused_blocks             number;
  7        l_unused_bytes              number;
  8        l_LastUsedExtFileId         number;
  9        l_LastUsedExtBlockId        number;
 10       l_LAST_OSED_BLOCK           number;
 11
 12      procedure get_data
 13      is
 14      begin
 15           dbms_space.free_blocks
 16            (segment_owner    => USER,
 17              segment_name    =>  'T',
 18              segment_type    =>  'TABLE',
 19              FREELIST_group_id => 0,
 20               free_blks           => l_free_blks);
 21
 22           dbms_space.unused_space
 23            (segment_owner    => USER,
```

```
24              segment_name   => 'T',
25              segment_type   => 'TABLE',
26              total_blocks        => i_total_blocks,
27              total_bytes         => l_total_bytes,
28              unused_blocks  => l_unused_blocks,
29        unused_bytes         => l_unused_bytes,
30              IAST_USED_EXTENT_FILE_ID => l_LastOsedExtFileId,
31              IAST_USED_EXTENT_BLOCK_ID => l_LastUsedExtBlockId,
32             LAST_USED_BLOCK => l_last_used_block);
33
34
35        dbms_output.put_line(L_free_blks || ' on FREELIST, ' ||
36                          to_char(1 total_blocks-l unused_blocks-l)  ||
37                                  ' used by table') ;
38       end;
39  begin
40      for i in 0 .. 10
41       loop
42          dbms_output.put('insert ' || to_char(i,'00') || ' ') ;
43            get_data;
44           insert into t (x) values (i) ;
45            commit;
46       end loop;
47
48
49      for i in 0 .. 10
50       loop
51          dbms_output.put('update ' || to_char(i,'00') || ' ') ;
52            get_data;
53            update t set y = null where x = i;
54            commit;
55       end loop;
56  end;
57  /

Procedure created.
```

Здесь мы используем две подпрограммы пакета **DBMS_SPACE,** информирующие о том, сколько блоков находится в списке **FREELIST** сегмента, сколько — выделено таблице, сколько — не используется и т.д. С помощью этой информации можно определить, сколько используемых таблицей блоков (до отметки максимального уровня таблицы) находятся в списке FREELIST. Затем я вставляю в таблицу 10 строк с непустым значением в столбце Y. После этого построчно изменяю столбец Y так, чтобы он имел значение **Null.** С учетом того, что размер блока базы данных — 8 Кбайт, при стандартных значениях PCTFREE — 10 процентов **и PCTUSED** — 40 процентов можно ожидать, что семь строк свободно поместятся в блок (представленный ниже расчет сделан без учета служебных областей в блоке/строке):

```
(2+1) байт для X + (1000+2) байт для Y = 1005
1005 байт/строку * 7 строк = 7035
8192 - 7035 байт (размер блока) = 1157 байт

Остается 1157 байт, что недостаточно для еще одной строки плюс чуть больше
800 байт (10% блока)
```

Поскольку 10 процентов от блока размером 8 Кбайт составляет немногим более 800 байт, понятно, что еще одна строка в блок не поместится. При желании можно точно рассчитать размер заголовка блока; здесь мы лишь предположили, что он поместится в 350 с небольшим байтах (1157 - 800 = 357). Это позволяет разместить в блоке семь строк.

Затем определим, сколько изменений понадобится, чтобы блок вернулся в список FREELIST. Мы знаем, что для этого блок должен использоваться менее чем на 40 процентов, т.е. для возврата в список свободных в блоке может использоваться не более 3275 байт. Тогда, если при выполнении каждого оператора UPDATE освобождается 1000 байт, для возвращения блока обратно в список FREELIST потребуется выполнить примерно четыре оператора UPDATE. Ну что же, давайте посмотрим, насколько хорошо я посчитал:

```
tkyte@TKYTE816> exec measure_usage;
insert   00  0 on FREELIST, 0 used by table
insert   01  1 on FREELIST, 1 used by table
insert   02  1 on FREELIST, 1 used by table
insert   03  1 on FREELIST, 1 used by table
insert   04  1 on FREELIST, 1 used by table
insert   05  1 on FREELIST, 1 used by table
insert   06  1 on FREELIST, 1 used by table
insert   07  1 on FREELIST, 1 used by table  — между 7-й к 8-й строками
insert   08  1 on FREELIST, 2 used by table    добавился еще один
insert   09  1 on FREELIST, 2 used by table    "используемый" блок
insert   10  1 on FREELIST, 2 used by table
update   00  1 on FREELIST, 2 used by table
update   01  1 on FREELIST, 2 used by table
update   02  1 on FREELIST, 2 used by table
update   03  1 on FREELIST, 2 used by table
update   04  2 on FREELIST, 2 used by table  — четвертое изменение
update   05  2 on FREELIST, 2 used by table    возвращает блок в список
update   06  2 on FREELIST, 2 used by table    свободных
update   07  2 on FREELIST, 2 used by table
update   08  2 on FREELIST, 2 used by table
update   09  2 on FREELIST, 2 used by table
update   10  2 on FREELIST, 2 used by table

PL/SQL procedure successfully completed.
```

Понятно, что после семи вставок в таблицу добавляется еще один блок. Аналогично, после четырех изменений, количество блоков в списке FREELIST увеличивается с 1 до 2 (оба блока опять находятся в списке свободных — в них можно вставлять строки). Если удалить и пересоздать таблицу T с другими установками и выполнить измерение еще раз, мы получим следующее:

```
tkyte@TKYTE816> create table t (x int, y char(1000) default 'x') pctfree 10
  2    pctused 80;

Table created.

tkyte@TKYTE816> exec measure_usage;
insert   00  0 on FREELIST,  0 used by table
insert   01  1 on FREELIST,  1 used by table
insert   02  1 on FREELIST,  1 used by table
insert   03  1 on FREELIST,  1 used by table
insert   04  1 on FREELIST,  1 used by table
insert   05  1 on FREELIST,  1 used by table
insert   06  1 on FREELIST,  1 used by table
insert   07  1 on FREELIST,  1 used by table
insert   08  1 on FREELIST,  2 used by table
insert   09  1 on FREELIST,  2 used by table
insert   10  1 on FREELIST,  2 used by table
update   00  1 on FREELIST,  2 used by table
update   01  2 on FREELIST,  2 used by table  - первое же изменение вернуло
update   02  2 on FREELIST,  2 used by table    блок в список свободных из-за
update   03  2 on FREELIST,  2 used by table    намного большего значения
update   04  2 on FREELIST,  2 used by table    pctused
update   05  2 on FREELIST,  2 used by table
update   06  2 on FREELIST,  2 used by table
update   07  2 on FREELIST,  2 used by table
update   08  2 on FREELIST,  2 used by table
update   09  2 on FREELIST,  2 used by table
update   10  2 on FREELIST,  2 used by table

PL/SQL procedure successfully completed.
```

Это демонстрирует последствия увеличения значения параметра **PCTUSED.** Первое же изменение вернуло блок в список **FREELIST.** Этот блок может использоваться для вставки новой строки намного быстрее.

Значит ли это, что необходимо увеличивать значение параметра **PCTUSED?** Нет, не обязательно. Это зависит от того, что впоследствии происходит с данными. Пусть приложение работает по следующему циклу:

1. Добавление данных (множество вставок).
2. Изменение данных, которое приводит к увеличению и уменьшению строк.
3. Снова добавление данных.

Тогда вообще **не нужно,** чтобы блоки возвращались в список свободных в результате изменения. Желательно установить параметру **PCTUSED** небольшое значение, чтобы блок попадал в список **FREELIST** только после удаления из него всех строк данных. В противном случае некоторые из блоков с временно уменьшенными строками при большом значении **PCTUSED** будут использованы для вставки новых строк. Затем, когда придется изменять старые и новые строки в этих блоках, для них не хватит места и придется часть строк переносить.

Итак, параметры **PCTUSED** и **PCTFREE имеют ключевое значение.** С одной стороны, их нужно задавать, чтобы избежать переноса слишком большого количества строк, а с другой, — чтобы избежать избыточных потерь пространства в блоках. Необходимо изучить объекты и их предполагаемое использование, а затем выработать обоснованный план установки значений этих параметров. Простые правила могут и не подойти; оптимальные значения действительно зависят от способов использования объектов. Имеет смысл рассмотреть следующие варианты (помните, большое и небольшое — понятия **относительные).**

- Большое значение **PCTFREE,** небольшое значение **PCTUSED.** Для вставки большого количества динамических данных, если при их изменении длина строк часто увеличивается. При этом резервируется много свободного пространства в блоке после вставки (большое значение **PCTFREE);** прежде чем блок будет возвращен в список свободных, он должен стать практически пустым (небольшое значение **PCTUSED).**
- Небольшое значение **PCTFREE,** большое значение PCTUSED. Если предполагается только вставка и удаление строк или если при изменении строки ее длина только уменьшается.

Параметры INITIAL, NEXT и PCTINCREASE

Эти параметры хранения определяют размер начального **(INITIAL)** и последующих экстентов, выделяемых таблице, а также процент увеличения для следующих экстентов. Например, если используется начальный экстент размером 1 Мбайт, следующий — размером 2 Мбайта, а параметр **PCTINCREASE** имеет значение 50, в таблице будут следующие экстенты:

1. 1 Мбайт.
2. 2 Мбайт.
3. 3 Мбайт (150 процентов от 2).
4. 4,5 Мбайт (150 процентов от 3).

и т.д. Я считаю эти параметры **устаревшими.** В базе данных необходимо использовать только локально управляемые табличные пространства с одинаковыми экстентами. При этом размер первого экстента всегда совпадает с размером следующего, а параметр **PCTINCREASE** вообще не имеет смысла, — он лишь вызывает фрагментацию табличного пространства.

Если локально управляемые табличные пространства не используются, я рекомендую всегда устанавливать **INITIAL** = NEXT и **PCTINCREASE** = 0. Это позволяет эмулировать процесс выделения пространства, происходящий в локально управляемых табличных пространствах. Для всех объектов в табличном пространстве должна использоваться одинаковая стратегия выделения экстентов во избежание фрагментации.

Параметры MINEXTENTS и MAXEXTENTS

Эти параметры управляют количеством экстентов, которые могут быть выделены объекту. Значение **MINEXTENTS** указывает серверу Oracle, сколько экстентов выделять таблице первоначально. Например, в локально управляемом табличном пространстве с одинаковыми экстентами размером 1 Мбайт при установке параметру **MINEXTENTS** значения 10 таблице будет выделено 10 Мбайт дискового пространства.

Параметр **MAXEXTENTS** задает верхний предел для количества экстентов, которые могут быть выделены объекту. Если в том же табличном пространстве задать параметру **MAXEXTENTS** значение 255, таблица сможет иметь размер не более 255 Мбайт. Конечно, если в табличном пространстве не хватит места для обеспечения такого роста, выделить соответствующие экстенты таблице тоже не удастся.

Параметры LOGGING и NOLOGGING

Обычно объекты создаются с журнализацией **(LOGGING),** тем самым все действия с объектом, которые могут генерировать информацию повторного выполнения, ее генерируют. Параметр **NOLOGGING** позволяет выполнять с объектом ряд действий, не генерируя данных повторного выполнения. Параметр **NOLOGGING** затрагивает лишь несколько специфических действий: первоначальное создание объекта, непосредственная загрузка данных с помощью утилиты **SQLLDR** или пересоздание (подробнее о том, какие из этих действий применимы к тому или иному объекту базы данных, см. в справочном руководстве по языку SQL — *"SQL Language Reference Manual").*

Этот параметр не отключает генерирование данных повторного выполнения для объекта вообще, а только для некоторых специфических действий. Например, если таблица создается как **SELECT NOLOGGING,** а затем выполняется оператор **INSERT INTO THAT_TABLE VALUES (1),** этот оператор будет зарегистрирован в журнале, а создание таблицы — нет.

Параметры INITRANS and MAXTRANS

Каждый блок данных объекта включает заголовок. Часть этого заголовка — таблица транзакций. В таблицу транзакций вносятся записи о том, какие строки/элементы блока заблокированы какими транзакциями. Первоначальный размер таблицы транзакций определяется параметром **INITRANS** объекта. Для таблиц стандартное значение — 1 (для индексов — 2). При необходимости таблица транзакций может динамически расти вплоть до **MAXTRANS** записей (естественно, при наличии достаточного свободного пространства в блоке). Каждая запись транзакции занимает 23 байта в заголовке блока.

Таблицы, организованные в виде кучи

Таблицы, организованные в виде кучи, используются приложениями в 99 (если не более) процентах случаев, хотя со временем это может измениться за счет более интенсивного использования таблиц, организованных по индексу, — ведь по таким таблицам

теперь тоже можно создавать дополнительные индексы. Таблица, организованная в виде кучи, создается по умолчанию при выполнении оператора CREATE TABLE. Если необходимо создать таблицу другого типа, это надо явно указать в операторе CREATE.

"Куча" — классическая структура данных, изучаемая в курсах программирования. Это по сути большая область пространства на диске или в памяти (в случае таблицы базы данных, конечно же, на диске), используемая произвольным образом. Данные размещаются там, где для них найдется место, а не в определенном порядке. Многие полагают, что данные будут получены из таблицы в том же порядке, в каком туда записывались, но при организации в виде кучи это не гарантировано. Фактически гарантировано как раз обратное: строки будут возвращаться в абсолютно непредсказуемом порядке. Это очень легко продемонстрировать. Создадим такую таблицу, чтобы в моей базе данных в блоке помещалась одна полная строка (я использую блоки размером 8 Кбайт). Совсем не обязательно создавать пример с одной строкой в блоке. Я просто хочу продемонстрировать предсказуемую последовательность событий. Такое поведение будет наблюдаться для таблиц любых размеров и в базах данных с любым размером блока:

```
tkyte@TKYTE816> create table t
  2  (a int,
  3  b varchar2(4000) default rpad('*',4000,'*'),
  4  c varchar2(3000) default rpad('*',3000,'*')
  5  )
  6  /

Table created.

tkyte@TKYTE816> insert into t (a) values (1);
1 row created.

tkyte@TKYTE816> insert into t (a) values (2);

1 row created.

tkyte@TKYTE816> insert into t (a) values (3);

1 row created.

tkyte@TKYTE816> delete from t where a = 2 ;

1 row deleted.

tkyte@TKYTE816> insert into t (a) values (4);

1 row created.

tkyte@TKYTE816> select a from t;

     A

     1
     4
     3
```

Если вы хотите воспроизвести этот пример, измените столбцы B и C в соответствии с размером блока. Например, при использовании блоков размером 2 Кбайт, столбец C

не нужен, а столбец B должен быть типа **VARCHAR2(1500)** со стандартным значением 1500 звездочек. Поскольку данные в этой таблице организованы в виде кучи, при появлении свободного пространства оно используется повторно. При полном просмотре данные выдаются в том порядке, в котором обнаруживаются, а не в порядке вставки. Это принципиально важное свойство обычных таблиц базы данных; в общем случае они представляют собой неупорядоченные наборы данных. Учтите, что для получения подобного результата необязательно использовать оператор **DELETE** — тех же результатов можно достичь с помощью только операторов **INSERT.** Если вставить маленькую строку, потом — очень большую, которая не поместится в один блок с маленькой строкой, а затем — опять маленькую строку, при выборке эти строки вполне могут быть получены в порядке "маленькая, маленькая, большая". Они не будут извлекаться в порядке вставки. Сервер Oracle размещает данные там, где они помещаются, а не в порядке выполнения транзакций.

Если в результате запроса данные должны выдаваться в порядке вставки, придется добавить столбец, который будет использоваться для упорядочения данных при извлечении. Этот столбец может быть числовым, например, с последовательно увеличивающимися значениями (которые генерируются с помощью объекта **SEQUENCE** базы данных Oracle). Затем можно будет примерно воспроизвести последовательность вставки, упорядочивая данные по этому столбцу. Это будет лишь приближение, поскольку строка с последовательным номером 55 вполне могла быть зафиксирована в базе данных до строки с последовательным номером 54, поэтому официально она была в базе данных "первой".

Поэтому таблицу, организованную в виде кучи, можно рассматривать просто как большой неупорядоченный набор строк. Эти строки будут выдаваться во внешне случайном порядке, зависящем от используемых опций (параллельные запросы, различные режимы оптимизации и т.д.), причем могут выдаваться в разном порядке даже для одного и того же запроса. Никогда не полагайтесь на порядок строк в результатах запроса, если только запрос не включает конструкцию **ORDER BY!**

Что еще важно знать о таблицах, организованных в виде кучи? Вообще-то, синтаксису оператора **CREATE TABLE** посвящено более 40 страниц в справочном руководстве по языку SQL, предлагаемом корпорацией Oracle, так что различных опций много. Опций настолько много, что разобраться со всеми весьма сложно. Одни только синтаксические ("рельсовые") схемы занимают восемь страниц. Чтобы описать поддерживаемые опции оператора **CREATE TABLE,** я создам максимально простую таблицу, например:

```
tkyte@TKYTE816> create table t
  2    (x int primary key ,
  3       y date,
  4       z clob)
  5    /

Table created.
```

Затем с помощью стандартных утилит экспорта и импорта данных (им посвящена глава 8), экспортирую определение таблицы и при импорте потребую показать полный текст оператора создания таблицы:

```
exp userid=tkyte/tkyte tables=t
imp userid=tkyte/tkyte full=y indexfile=t.sql
```

В результате можно обнаружить, что в файле T.SQL содержится максимально подробный оператор создания соответствующей таблицы. Я его немного переформатировал для простоты чтения, но текст взят из файла, сгенерированного утилитой экспорта:

```
CREATE TABLE "TKYTE"."T"
 ("X" NUMBER(*,0), "Y" DATE, "Z" CLOB)
PCTFREE 10 PCTUSED 40
INITRANS 1 MAXTRANS 255
LOGGING STORAGE(INITIAL 32768 NEXT 32768
                MINEXTENTS 1 MAXEXTENTS 4096
                PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
                BUFFER_POOL DEFAULT
                )
TABLESPACE "TOOLS"
LOB ("Z") STORE AS (TABLESPACE "TOOLS"
                    ENABLE STORAGE IN ROW CHUNK 8192
                    PCTVERSION 10 NOCACHE
                    STORAGE(INITIAL 32768 NEXT 32768
                            MINEXTENTS 1 MAXEXTENTS 4096
                            PCTINCREASE 0
                            FREELISTS 1 FREELIST GROUPS 1
                            BUFFER_POOL DEFAULT)) ;
ALTER TABLE "TKYTE"."T"
ADD PRIMARY KEY ("X")
USING INDEX
PCTFREE 10 INITRANS 2 MAXTRANS 255
STORAGE(INITIAL 32768 NEXT 32768
        MINEXTENTS 1 MAXEXTENTS 4096
        PCTINCREASE 0
        FREELISTS 1 FREELIST GROUPS 1
        BUFFER_POOL DEFAULT)
TABLESPACE "TOOLS" ENABLE ;
```

В этом тексте показаны многие опции оператора CREATE TABLE. Мне достаточно было только указать типы данных и имена столбцов, а детальную версию сервер Oracle выдал автоматически. Теперь можно отредактировать эту детальную версию, заменив, например, конструкцию ENABLE STORAGE IN ROW конструкцией DISABLE STORAGE IN ROW. Это отключит хранение данных больших объектов в строке вместе со структурированными данными и приведет к созданию отдельного сегмента. Я использую этот прием постоянно, чтобы сэкономить время на обдумывание больших синтаксических схем. Он также позволяет понять, какие опции оператора CREATE TABLE доступны в разных ситуациях.

Так я определяю, что же можно синтаксически задать в операторе **CREATE TABLE. Я** использую этот прием для многих объектов: создаю небольшую тестовую схему, в ней — простейшие объекты, экспортирую ее с помощью параметра OWNER = ЭТА_СХЕМА и выполняю импорт. Просматривая сгенерированный при импорте SQL-файл, я определяю, какие опции доступны.

Теперь, зная, как увидеть большинство имеющихся опций оператора CREATE TABLE, попробуем разобраться, какие из них наиболее важны для таблиц, организованных **в виде кучи? По** моему мнению, это следующие опции.

- **FREELISTS.** Для каждой таблицы выделенные из кучи блоки отслеживаются в списке свободных, **FREELIST.** Таблица может иметь несколько списков **FREELIST.** Если предполагается интенсивная вставка данных в таблицу большим количеством пользователей, создание нескольких списков **FREELIST** может существенно повысить производительность (за счет использования дополнительного пространства). Влияние значения этого параметра на производительность рассматривалось ранее (в разделе "Списки свободных мест").
- PCTFREE. Степень заполнения блока в процессе вставки строк. Как только в блоке осталось менее чем **PCTFREE** процентов свободного пространства, он уже не просматривается при вставке новых строк. Этот параметр позволяет контролировать перенос строк при последующих изменениях и устанавливается в зависимости от предполагаемого использования таблицы.
- **PCTUSED.** Указывает, насколько пустым должен стать блок, чтобы в него можно было вставлять строки. Новые строки будут вставляться в блок, только если он занят менее чем на **PCTUSED** процентов. Как и в случае параметра **PCTFREE,** для правильной установки значения этого параметра необходимо учитывать предполагаемое использование таблицы.
- **INITRANS.** Количество записей в таблице транзакций, первоначально выделяемое в блоке. При слишком низком значении (стандартное значение — 1) могут возникнуть проблемы при одновременном доступе большого количества пользователей. Если блок базы данных почти заполнен и таблицу транзакций нельзя динамически увеличить, сеансы будут ожидать в очереди доступа к блоку, поскольку для каждой из одновременно выполняемых транзакций необходима запись в таблице транзакций. Если предполагается одновременное выполнение большого количества изменений одних и тех же блоков, имеет смысл увеличить значение этого параметра.

Примечание: для данных больших объектов, хранящихся в отдельном сегменте вне строки, не используются значения параметров **PCTFREE/PCTUSED,** установленные для таблицы. Эти блоки больших объектов управляются по-другому. Они всегда заполняются полностью и возвращаются в список свободных, когда полностью пусты.

Именно на эти параметры надо обращать особое внимание. Я обнаружил, что остальные параметры хранения теперь просто не слишком важны. Как я уже упоминал ранее в этой главе, необходимо использовать локально управляемые табличные пространства, а для них параметры **PCTINCREASE, NEXT** и т.п. не используются.

Таблицы, организованные по индексу

Таблицы, организованные по индексу (index organized tables — IOTs), — это таблицы, хранящиеся в структуре индекса. Таблица, хранящаяся в куче, организована случайным образом (данные попадают в любое свободное место). В таблице же, организованной по индексу, хранимые данные отсортированы по первичному ключу. С точки зрения приложений, таблицы, организованные по индексу, ничем не отличаются: к ним применяются такие же SQL-операторы, как и для доступа к обычной таблице. Они особенно полезны для информационно-поисковых (information retrieval — IR) систем, хранения пространственных данных и приложений оперативного анализа информации (OLAP).

Зачем организовывать таблицу по индексу? Можно задать и прямо противоположный вопрос; зачем организовывать таблицу в виде кучи? Поскольку все таблицы в реляционной базе данных в любом случае должны иметь первичный ключ, не будет ли организация таблицы в виде кучи непозволительной тратой пространства? При использовании таблицы, организованной в виде кучи, необходимо управлять дисковым пространством как для таблицы, так и индекса по первичному ключу. При использовании таблицы, организованной по индексу, дополнительное пространство для поддержки индекса по первичному ключу не требуется, поскольку индекс — это данные, а данные — это индекс. Проблема в том, что индекс — сложная структура данных, поддержка и управление которой требуют выполнения множества действий. Кучей же управлять сравнительно легко. В некоторых аспектах таблица, организованная в виде кучи, эффективнее таблицы, организованной по индексу. Тем не менее, у таблиц, организованных по индексу, есть ряд преимуществ. Например, однажды я создавал индекс на базе обратного списка (дело было еще до появления interMedia и соответствующих технологий). У меня была таблица документов. Я анализировал документы и находил в них слова. Еще у меня была создана таблица следующего вида:

```
create table keywords (
word varchar2(50),
position int,
doc_id int,
primary key(word,position,doo_id));
```

Эта таблица состояла исключительно из столбцов первичного ключа. При этом расходовалось двойное количество пространства; размеры таблицы и индекса по первичному ключу были сопоставимы (на самом деле индекс по первичному ключу был больше, поскольку в нем хранился идентификатор строки, на которую указывал ключ, а в таблице идентификатор строки отдельно не хранится — он определяется). Я применял к этой таблице SQL-операторы только с конструкцией WHERE, задающей условие для столбца WORD или столбцов WORD и POSITION. Другими словами, я никогда не использовал таблицу, а только индекс таблицы. На таблицу только попусту расходовалось дисковое пространство. Мне нужно было находить все документы, содержащие заданное слово (или одно слово "рядом" с другим и т.п.). Таблица была бесполезна, она просто замедляла работу приложения и удваивала требования к дисковому пространству. Такие таблицы сам бог велел организовывать по индексу.

Еще одна таблица, так и напрашивающаяся на организацию по индексу, — справочник с поиском по коду. Предположим, необходимо получать название штата по почтовому индексу. Для этого подходит только таблица, организованная по индексу. Любую таблицу, к которой часто обращаются по первичному ключу, лучше всего организовывать по индексу.

Организация таблицы по индексу пригодится при реализации собственной индексной структуры. Предположим, необходимо обеспечить для приложения поиск строки, независимо от регистра символов. Для этого можно использовать индексы по функции (подробнее о них см. в главе 7). Однако такие индексы поддерживаются только в версиях Oracle Enterprise и Personal Edition. Если имеется версия Standard Edition, одним из способов обеспечения поиска ключевых слов независимо от регистра будет создание собственного "индекса по функции". Пусть, например, необходимо обеспечить не зависящий от регистра поиск по столбцу **ENAME** в таблице **EMP.** Один из способов — создать еще один столбец, **ENAME_UPPER,** в таблице **EMP** и проиндексировать его. Этот "теневой" столбец будет поддерживаться с помощью триггера. Если идея добавления столбца в таблицу вам не нравится, можно просто создать собственный индекс по функции следующим образом:

```
tkyte@TKYTE816> create table emp as select * from scott.emp;

Table created.

tkyte@TKYTE816>create table upper_ename
  2  (x$ename,  x$rid,
  3     primary key  (x$ename,x$rid)
  4  )
  5  organization index
  6  as
  7  select upper(ename), rowid from emp
  8  /
Table created.

tkyte@TKYTE816> create or replace trigger upper_ename
  2  after insert or update or delete on emp
  3  for each row
  4  begin
  5       if  (updating and  (:old.ename||'x'  <>  :new.ename||'x'))
  6       then
  7            delete  from upper_ename
  8             where x$ename = upper(:old.ename)
  9                and x$rid =  :old.rowid;
 10
 11            insert into upper_ename
 12            (x$ename,x$rid)  values
 13            (upper(:new.ename),  :new.rowid);
 14       elsif  (inserting)
 15       then
 16            insert into upper_ename
```

```
 17            (x$ename,x$rid)  values
 18            (upper(:new.ename),  :new.rowid);
 19       elsif  (deleting)
 20       then
 21            delete  from upper_ename
 22             where  x$ename = upper(:old.ename)
 23                and x$rid =  :old.rowid;
 24       end if;
 25  end;
 26  /

Trigger created.

tkyte@TKYTE816> update emp set ename - initcap(ename);

14 rows updated.

tkyte@TKYTE816> commit;

Commit complete.
```

Итак, таблица UPPER_ENAME фактически представляет собой индекс, не зависящий от регистра символов, во многом аналогичный индексу по функции. Этот "индекс" надо использовать явно — серверу Oracle о нем не известно. Следующие примеры показывают, как можно использовать этот "индекс" для изменения, выборки и удаления данных из таблицы.

```
tkyte@TKYTE816> update
  2   (
  3   select ename,  sal
  4     from emp
  5    where emp.rowid in  (select upper_ename.x$rid
  6                           from upper_ename
  7                          where x$ename =  'KING')
  8   )
  9   set sal  = 1234
 10   /

1 row updated.
tkyte@TKYTE816> select ename,  empno,  sal
  2     from emp,  upper_ename
  3    where emp.rowid = upper_ename.x$rid
  4      and upper_ename.x$ename =  'KING'
  5   /

ENAME               EMPNO          SAL

King                 7839         1234

tkyte@TKYTE816> delete  from
  2   (
  3   select ename,  empno
  4     from emp
```

```
  5    where emp.rowid in  (select uppex_ename.x$rid
  6                           from upper_ename
  7                          where x$ename =  'KINS'}
  8  )
  9  /

1 row deleted.
```

При выборе можно использовать подзапрос с IN либо соединение (JOIN). Из-за правил "сохранения ключей" при изменении или удалении строки надо использовать только подзапрос с IN. Примечание об этом методе, требующем сохранения идентификаторов строк: организованную по индексу таблицу, как и любой другой индекс, необходимо перестраивать, если в результате выполненных действий, таких как экспорт и импорт или применение оператора ALTER TABLE MOVE, изменились идентификаторы строк в таблице EMP.

Наконец, если необходимо обеспечить совместное размещение данных или физически хранить данные в определенном порядке, индексная организация таблицы тоже подойдет. Пользователи СУБД Sybase и SQL Server в этом случае использовали бы кластерный индекс, но организация таблицы по индексу намного лучше. В случае кластерного индекса в этих СУБД может дополнительно расходоваться до 110 процентов пространства (аналогично моей таблице KEYWORDS в представленном ранее примере). Здесь же дополнительных расходов вообще нет, поскольку данные хранятся только в одном месте. Классический пример желательности совместного размещения взаимосвязанных данных представляет собой отношение главный/подчиненный. Пусть у таблицы EMP имеется подчиненная таблица:

```
tkyte@TKYTE816> create table addresses
  2   (empno       number(4) references emp(empno) on delete cascade,
  3    addr_type   varchar2(10) ,
  4    street      varchar2(20) ,
  5    city        varchar2(20) ,
  6    state       varchar2(2),
  7    zip         number,
  8    primary key (empno,addr_type)
  9  )
 10  ORGANIZATION INDEX
 11  /

Table created.
```

Физически близкое размещение всех адресов сотрудника (домашнего адреса, адреса места работы, адреса школы, прежнего адреса и т.д.) уменьшит объем ввода/вывода при соединении таблиц EMP и ADDRESSES. Объем логического ввода/вывода будет тем же, но физического ввода/вывода потребуется существенно меньше. В обычной таблице каждый адрес сотрудника может физически находиться в другом блоке по отношению к другим адресам того же сотрудника. Сортируя адреса по столбцам EMPNO и ADDR_TYPE, мы гарантируем, что все адреса данного сотрудника хранятся "рядом".

Это применимо и в случае частого использования запросов с конструкцией BETWEEN по первичному или уникальному ключу. Хранение отсортированных данных повысит производительность и этих запросов. Например, в моей базе данных имеется таблица котировок акций. Каждый день в нее собирается информация о символах акций, дате, конечной цене, колебаниях курса в течение дня, количестве проданных акций и т.д. Это делается для акций сотен компаний. Соответствующая таблица имеет такой вид:

```
tkyte@TKYTE816> create table stocks
  2  (ticker    varchar2(10),
  3   day       date,
  4   value     number,
  5   change    number,
  6   high      number,
  7   low       number,
  8   vol       number,
  9   primary key(ticker,day)
 10  )
 11  organization index
 12  /

Table created.
```

Я часто просматриваю котировки акций одной компании, за некоторый период (вычисляя скользящее среднее, например). При использовании таблицы, организованной в виде кучи, вероятность того, что две строки для акций ORCL окажутся в одном блоке базы данных, практически равна нулю. Дело в том, что каждую ночь вставляются записи за день для всех акций. При этом заполняется как минимум один блок базы данных (обычно — много блоков). Поэтому каждый день мы добавляем новую запись для акций ORCL, но она попадает в блок, не совпадающий с другими, содержащими записи для ORCL, блоками таблицы. Если выполнить запрос:

```
Select * from stocks
 where ticker = 'ORCL'
   and day between sysdate and sysdate - 100;
```

сервер Oracle прочитает индекс, а затем обратится к таблице за остальными данными строки по идентификатору строки. Каждая из 100 выбираемых строк окажется в другом блоке базы данных из-за принятого способа загрузки данных в таблицу, и каждая, вероятно, потребует выполнения операции физического ввода/вывода. Теперь предположим, что такая таблица организована по индексу. При выполнении этого же запроса придется прочитать только соответствующие блоки индекса, и все необходимые данные будут получены. Не только нет необходимости обращаться к таблице, но и все строки для акций ORCL в заданном диапазоне дат физически хранятся "рядом" друг с другом. Требуется меньше логического и физического ввода/вывода.

Теперь понятно, когда следует использовать таблицы, организованные по индексу, и как это делать. Осталось разобраться, какие опции можно использовать при создании таких таблиц? Есть ли потенциальные проблемы при их использовании? Опции те же, что задаются для таблиц, организованных в виде кучи. Еще раз воспользуемся утилита-

ми EXP/IMP, чтобы увидеть детали. Если начать с трех простейших разновидностей таблицы, организованной по индексу:

```
tkyte@TKYTE816> create table t1
  2   (x int primary key,
  3    y varchar2(25),
  4    z date
  5   )
  6   organization index;

Table created.

tkyte@TKYTE816> create table t2
  2   (x int primary key,
  3    y varchar2(25),
  4    z date
  5   )
  6   organization index
  7   OVERFLOW;
Table created.

tkyte@TKYTE816> create table t3
  2   (x int primary key,
  3    y varchar2(25),
  4    z date
  5   )
  6   organization index
  7   overflow INCLUDING y;

Table created.
```

Прежде чем обсуждать ключевые слова OVERFLOW и INCLUDING, давайте сначала рассмотрим полный текст SQL-оператора, необходимого для создания первой из представленных выше таблиц:

```
CREATE TABLE "TKYTE"."T1"
 ("X" NUMBER(*,0) ,
  "Y" VARCHAR2(25) ,
  "Z" DATE,
  PRIMARY KEY  ("X")  ENABLE
)
ORGANIZATION INDEX
NOCOMPRESS
PCTFREE 10
INITRANS 2 MAXTRANS 255
LOGGING
STORAGE  ( INITIAL 32768
             NEXT 32768
             MINEXTENTS 1 MAXEXTENTS 4096
             PCTINCREASE 0
             FREELISTS 1
```

```
            FREELIST GROUPS 1
            BUFFER_POOL DEFAULT
          )
TABLESPACE "TOOLS"
PCTTHRESHOLD 50 ;
```

Добавились две новые опции — NOCOMPRESS и PCTTHRESHOLD. Мы их вскоре рассмотрим. Возможно, вы обратили внимание, что в тексте оператора CREATE TABLE чего-то не хватает; конструкции PCTUSED нет, а PCTFREE — есть. Это потому, что индекс — сложная структура данных, организованная не случайным образом, как куча, а так, чтобы данные попали в определенное место. В отличие от кучи, где блоки доступны для вставки время от времени, блоки индекса всегда доступны для вставки новых записей. Если данные принадлежат определенному блоку в соответствии со значением ключа, они попадут в него независимо от степени его заполнения. Кроме того, параметр PCTFREE используется только при создании объекта и наполнении данными индексной структуры. В таблицах, организованных в виде кучи, он используется иначе. Параметр PCTFREE резервирует пространство во вновь созданном индексе, но не для последующих операций с ним (во многом, по той же причине, почему вообще не используется параметр PCTUSED). Все соображения относительно списков свободных блоков FREELIST, высказанные для таблиц, организованных в виде кучи, относятся и к таблицам, организованным по индексу.

Теперь перейдем к новой опции NOCOMPRESS. Эта опция используется для индексов. Она требует, чтобы сервер Oracle хранил все значения в записи индекса (не сжимал ее). Если первичный ключ объекта создан по столбцам A, B и C, будут храниться все комбинации A, B и C. Противоположностью NOCOMPRESS является опция COMPRESS N, где N — целое число, задающее количество сжимаемых столбцов. В результате удаляются повторяющиеся значения; они факторизуются на уровне блоков, так что повторяющиеся значения столбца A и, возможно, B, больше не хранятся. Рассмотрим пример таблицы:

```
tkyte@TKYTE816> create table iot
  2   (owner, object_type, object_name,
  3    primary key(owner,object_type,object_name)
  4   )
  5   organization index
  6   NOCOMPRESS
  7   as
  8   select owner, object_type, object_name from all_objects
  9   /

Table created.
```

Значение в столбце OWNER повторяется много сотен раз. Каждой схеме (OWNER) обычно принадлежит множество объектов. Даже пара значений OWNER, OBJECT_TYPE повторяется многократно; в схеме имеются десятки таблиц, десятки пакетов и т.д. Не повторяются только все три столбца вместе. Можно попросить сервер Oracle убрать повторяющиеся значения. Вместо создания индексного блока со значениями:

Sys,table,t1	Sys,table.t2	Sys.table,t3	Sys,table,t4
Sys,table,t5	Sys,table,t6	Sys,table,t7	Sys,table,t8
...	...	...	...
Sys,table,t100	Sys,table,t101	Sys,table,t102	Sys,table,t103

можно использовать конструкцию **COMPRESS** 2 (факторизовать первых два столбца) и получить блок с такими данными:

Sys.table	t1	t2	t3
t4	t5 ...	..	
...	t103	t104	..
t300	t301	t302	t303

Т.е. значения SYS и TABLE сохраняются лишь один раз, а затем — только значения третьего столбца. При этом в блок индекса может поместиться намного больше записей. При этом ни степень параллелизма, ни функциональные возможности никак не ограничиваются. Требуется чуть больше процессорного времени, поскольку сервер Oracle выполняет дополнительные действия, чтобы собрать значения ключей. Однако при этом существенно сокращается объем данных при вводе/выводе, а в буферном кэше помещается больше данных, поскольку их больше помещается в блоке. Это очень хороший компромисс. Мы продемонстрируем экономию с помощью простого тестового примера, в котором создадим представленную ранее таблицу **(CREATE TABLE AS SELECT)** с параметрами **NOCOMPRESS, COMPRESS 1** и **COMPRESS 2.** Начнем с процедуры, показывающей использование пространства таблицей, организованной по индексу:

```
tkyte@TKYTE816> create or replace
  2  procedure show_iot_space
  3   (p_segname in varchar2)
  4   as
  5        l_segname       varchar2(30);
  6        l_total_blocks        number;
  7        l_total_bytes         number;
  8        l_unused_blocks       number;
  9        l_unused_bytes        number;
 10       l_LastUsedExtFileId   number;
 11     l_LastUsedExtBlockId    number;
 12       l_last_used_block     number;
 13  begin
 14       select 'SYS_IOT_TOP_' || object_id
 15         into l_segname
 16         from user_objects
 17        where object_name = upper(p_segname);
 18
 19       dbms_space.unused_space
 20        (segment_owner        => user,
 21         segment_name         => l_segname,
```

```
 22         segment_type          =>  'INDEX',
 23         total_blocks          =>  l_total_blocks,
 24          total_bytes          =>  l_total_bytes,
 25         unused_blocks         =>  l_unused_blocks,
 26         unused_bytes          =>  l_unused_bytes,
 27         LAST_USED_EXTENT_FILE_ID  =>  l_LastUsedExtFileId,
 28         LAST_USED_EXTENT_BLOCK_ID  =>  l_LastOsedExtBlockId,
 29         LAST_USED_BLOCK       =>  l_last_used_block) ;
 30
 31       dbms_output.put_line
 32       ("IOT used '  ||  to_char(l_total_blocks-l_unused_blocks));
 33  end;
 34  /

Procedure created.
```

Теперь создадим организованную по индексу таблицу, не используя сжатие:

```
tkyte@TKYTE816>  create  table  lot
   2   (owner,  object_type,  object_name,
   3    primary  key(owner,object_type,object_name)
   4   )
   5   organization  index
   6   NOCOMPRESS
   7   as
   8   select  owner,  object_type,  object_name  from  all_objects
   9   order  by  owner,  object_type,  object_name
  10   /

Table  created.

tkyte@TKYTE816>  set  serveroutput  on
tkyte@TKYTE816>  exec  show_iot_space('iot') ;
IOT used 135

PL/SQL  procedure  successfully  completed.
```

Если вы проверяете эти примеры по ходу чтения, то скорее всего получите другое значение, не 135. Оно будет зависеть от размера блока и количества объектов в словаре данных. Можно, однако, предположить, что в следующем примере это значение будет меньше:

```
tkyte@TKYTE816>  create  table  iot
   2   (owner,  object_type,  object_name,
   3    primary  key(owner,object_type,object_name)
   4   )
   5   organization  index
   6   compress  1
   7   as
   8   select  owner,  object_type,  object_name  from  all_objects
   9   order  by  owner,  object_type,  object_name
  10   /
```

```
Table created.

tkyte@TKYTE816> exec show_iot_space('iot');
IOT used 119

PL/SQL procedure successfully completed.
```

Итак, эта организованная по индексу таблица примерно на 12 процентов меньше первой; но ее можно еще сжать:

```
tkyte@TKYTE816> create table iot
  2   (owner, object_type, object_name,
  3    primary key(owner,object_type,object_name)
  4   )
  5   organization index
  6   compress 2
  7   as
  8   select owner, object_type, object_name from all_objects
  9   order by owner, object_type, object_name
 10   /
Table created.

tkyte@TKYTE816> exec show_iot_space('iot') ;
IOT used 91

PL/SQL procedure successfully completed.
```

Организованная по индексу таблица с параметром COMPRESS 2 примерно на треть меньше, чем исходная, несжатая. У вас может получиться другое значение, но результаты бывают просто фантастические.

Этот пример дает интересную информацию о таблицах, организованных по индексу. Это — таблицы, но только по названию. Их сегмент — полноправный сегмент индекса. Чтобы продемонстрировать использование пространства, мне пришлось получить по имени таблицы, организованной по индексу, соответствующее имя базового индекса. В этих примерах я позволил системе автоматически сгенерировать имя индекса; по умолчанию получается имя **SYS_IOT_TOP_<идентификатор_объекта>,** где **идентификатор_объекта** — внутренний идентификатор, присвоенный таблице. Если не хочется, чтобы эти автоматически сгенерированные имена засоряли словарь данных, можно задать имя явно:

```
tkyte@TKYTE816> create table iot
  2   (owner, object_type, object_name,
  3    constraint iot_pk primary key(owner,object_type,object_name)
  4   )
  5   organization index
  6   compress 2
  7   as
  8   select owner, object_type, object_name from all_objects
  9   /

Table created.
```

Обычно считается правильным вот так, явно, именовать создаваемые объекты. Обычно такие имена намного **информативнее, чем имена вида SYS_IOT_TOP_1234.**

Я пока отложу описание параметра **PCTTHRESHOLD,** поскольку он связан со следующими двумя параметрами таблиц, организованных по индексу, — **OVERFLOW** и **INCLUDING.** Если посмотреть на текст полных SQL-операторов для таблиц T2 и T3, можно увидеть следующее:

```
CREATE TABLE "TKYTE"."T2"
 ("X" NUMBER(*,0),
  "Y" VARCHAR2(25),
  "Z" DATE,
  PRIMARY KEY  ("X")  ENABLE
)
ORGANIZATION  INDEX
NOCOMPRESS
PCTFREE  10
INITRANS  2  MAXTRANS  255
LOGGING
STORAGE  (  INITIAL  32768  NEXT  32768  MINEXTENTS  1  MAXEXTENTS  4096
            PCTINCREASE  0  FREELISTS  1  FREELIST GROUPS  1  BUFFER_POOL  DEFAULT
)
TABLESPACE  "TOOLS"
PCTTHRESHOLD  50
OVERFLOW
            PCTFREE  10
                          PCTUSED  40
            INITRANS  1
            MAXTRANS  255
            LOGGING
            STORAGE  (  INITIAL  32768  NEXT  32768  MINEXTENTS  1  MAXEXTENTS  4096
                        PCTINCREASE  0  FREELISTS  1  FREELIST GROUPS  1
                        BUFFER_POOL  DEFAULT  )
TABLESPACE  "TOOLS"  ;
CREATE  TABLE
"TKYTE"."T3"
 ("X"  NUMBER(*,0),
  "Y" VARCHAR2(25),
  "Z" DATE,
  PRIMARY KEY  ("X")  ENABLE
)
ORGANIZATION  INDEX
NOCOMPRESS
PCTFREE  10
INITRANS  2
MAXTRANS  255
LOGGING
STORAGE(INITIAL  32768  NEXT  32768  MINEXTENTS  1  MAXEXTENTS  4096
          PCTINCREASE  0  FREELISTS  1  FREELIST GROUPS  1  BUFFER_POOL  DEFAULT  )
TABLESPACE  "TOOLS"
PCTTHRESHOLD  50
```

```
INCLUDING "Y"
OVERFLOW PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 LOGGING
          STORAGE ( INITIAL 32768 NEXT 32768 MINEXTENTS 1 MAXEXTENTS 4096
                    PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
                    BUFFER_POOL DEFAULT )
TABLESPACE "TOOLS" ;
```

Итак, осталось разобраться с параметрами PCTTHRESHOLD, **OVERFLOW** и INCLUDING. Они взаимосвязаны и позволяют обеспечить более эффективное хранение данных в листовых блоках индекса (именно в этих блоках и хранятся фактические данные индекса). Индекс обычно создается по подмножеству столбцов таблицы. Обычно в блоке индекса помещается во много раз больше строк, чем в блоке таблицы, организованной в виде кучи. Эффективность индекса зависит от возможности хранить в блоке много строк; в противном случае серверу Oracle придется тратить много времени на поддержку структуры индекса, ведь каждый оператор INSERT или **UPDATE** при этом может приводить к делению индексного блока, чтобы он мог вместить новые данные.

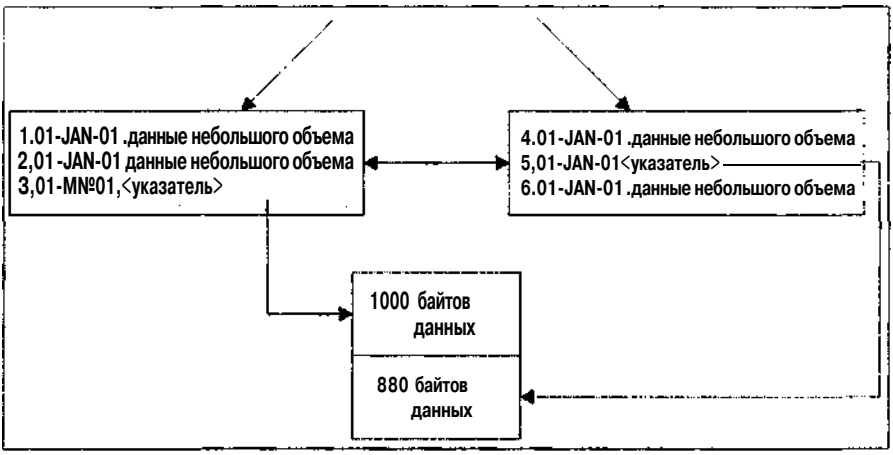
Конструкция OVERFLOW позволяет задать другой сегмент, дополнительный сегмент, в который помещаются данные строк организованной по индексу таблицы, когда они становятся слишком большими. Обратите внимание, что для сегмента **OVERFLOW** опять можно задавать параметр PCTUSED. Параметры PCTFREE и PCTUSED для сегмента OVERFLOW имеют такое же значение, что и для таблицы, организованной в виде кучи. Условия использования дополнительного сегмента можно задавать двумя способами.

- С помощью конструкции **PCTTHRESHOLD.** Когда объем данных в строке превзойдет этот процент от размера блока, хвостовые столбцы такой строки будут храниться в дополнительном сегменте. Итак, если параметр **PCTTHRESHOLD** имеет значение 10 процентов, а размер блока — 8 Кбайт, любая строка размером более 800 байт будет частично храниться в другом месте, вне блока индекса.
- С помощью конструкции INCLUDING. Все столбцы строки, вплоть до столбца, порядковый номер которого указан в конструкции INCLUDING, хранятся в блоке индекса, а остальные столбцы — в дополнительном сегменте.

Предположим, имеется таблица в базе данных с размером блока 2 Кбайт:

```
ops$tkyte@ORA8I.WORLD> create table iot
  2   (x     int,
  3    y     date,
  4    z     varchar2(2000) ,
  5    constraint iot_pk primary key (x)
  6   )
  7   organization index
  8   pctthreshold 10
  9   overflow
 10   /

Table created.
```

Графически ее можно представить так:

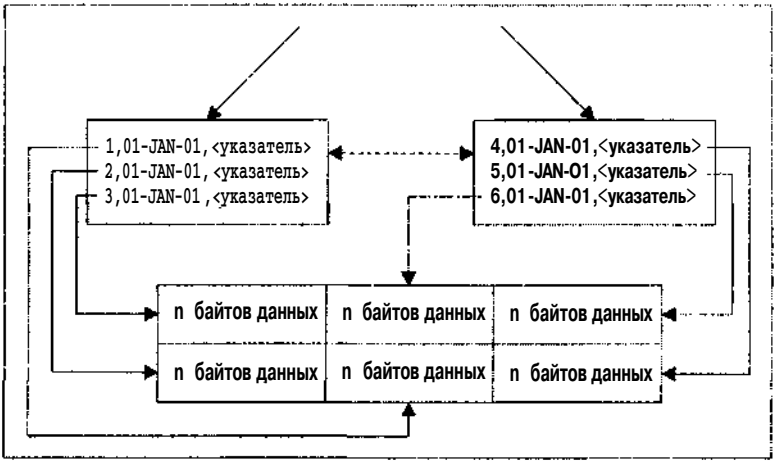


Серые прямоугольники — это записи индекса, часть большей индексной структуры (в главе 7, посвященной индексам, вы увидите более детальную схему структуры индекса). Если коротко, структура индекса — дерево, а листовые блоки (в которых хранятся данные) фактически образуют двухсвязный список, упрощающий последовательный просмотр блоков. Белый прямоугольник представляет дополнительный сегмент, в нем будут храниться данные, превосходящие устанавливаемый параметром PCTTHRESHOLD предел. Сервер Oracle будет просматривать столбцы в обратном порядке, начиная с последнего столбца строки и заканчивая последним столбцом первичного ключа (но не включая его), чтобы определить, какие столбцы надо хранить в дополнительном сегменте. В нашем примере числовой столбец X и столбец типа даты Y всегда будут помещаться в блоке индекса. Последний столбец, Z, имеет переменную длину. Когда его длина составляет менее 190 байт (10 процентов от блока размером 2 Кбайт — это около 200 байт, но надо вычесть 7 байт для даты и от 3 до 5 — для числа), он будет храниться в блоке индекса. Когда же его длина превысит 190 байт, сервер Oracle будет хранить данные столбца Z в дополнительном сегменте, и добавит указатель на них в блок индекса.

Еще можно использовать конструкцию INCLUDING. Она позволяет явно указать, какие столбцы должны храниться в блоке индекса, а какие — в дополнительном сегменте. Для следующей таблицы:

```
ops$tkyte@ORA8I.WORLD> create table lot
  2   (x     int,
  3    y     date,
  4    z     varchar2(2000),
  5    constraint iot_pk primary key  (x)
  6   )
  7   organization index
  8   including y
  9   overflow
 10   /

Table created.
```

можно ожидать такую структуру:



В этом случае столбец Z, независимо от объема содержащихся в нем данных, будет храниться "вне строки", в сегменте остатка.

Что же лучше использовать, **PCTTHRESHOLD, INCLUDING** или комбинацию обоих параметров? Это зависит от потребностей. Если имеется приложение, всегда или почти всегда использующее первые четыре столбца таблицы и лишь изредка обращающееся к последним пяти столбцам, имеет смысл применять конструкцию INCLUDING. В индекс включаются столбцы до четвертого включительно, а остальные пять будут храниться отдельно. Если в процессе работы они понадобятся, то будут выбраны аналогично перенесенным или фрагментированным строкам. Сервер Oracle прочитает начало строки, найдет указатель на остаток строки, а затем прочитает и его. Если же с определенностью нельзя сказать, что преимущественно обращаются именно к этим столбцам и лишь изредка — к остальным, имеет смысл подумать об использовании параметра **PCTTHRESHOLD.** Установить подходящее значение параметра **PCTTHRESHOLD** легко, если определить, сколько строк в среднем желательно хранить в блоке индекса. Предположим, необходимо хранить в индексном блоке 20 строк. Это означает, что каждая строка должна занимать 1/20 (5 процентов) блока. Параметр **PCTTHRESHOLD** должен иметь значение 5; каждый фрагмент строки, хранящийся в листовом блоке индекса, должен занимать не более 5 процентов блока.

И последнее, о чем следует поговорить, завершая рассмотрение таблиц, организованных по индексу, — индексация. Можно создавать индекс по индексу, если только первичный индекс представляет собой таблицу, организованную по индексу. Созданные таким образом индексы называются вторичными. Обычно индекс содержит физический адрес строки, на которую ссылается, в виде идентификатора строки. Вторичный индекс организованной по индексу таблицы не может использовать физические адреса; для ссылки на строку необходим другой способ. Дело в том, что строка в таблице, организованной по индексу, может часто перемещаться, причем она не переносится как строка в обычной таблице. Строка в таблице, организованной по индексу, должна находиться в определенном месте индексной структуры, которое определяется ее первичным клю-

чом; она будет перемещаться лишь при изменении размера и структуры самого индекса. Чтобы справиться с этой проблемой, сервер Oracle поддерживает логические идентификаторы строк. Эти логические идентификаторы строк основаны на первичном ключе таблицы, организованной по индексу. Они также могут включать "подсказку" о текущем местонахождении строки (хотя через некоторое время эта подсказка становится неверной, поскольку данные в организованной по индексу таблице перемещаются). Индекс по таблице, организованной по индексу, менее эффективен, чем по обычной таблице. Доступ по индексу к обычной таблице обычно требует ввода/вывода для просмотра структуры индекса, а затем одного чтения данных таблицы. В случае таблицы, организованной по индексу, обычно выполняется два просмотра: по структуре вторичного индекса и по самой таблице. Поэтому индексы по таблицам, организованным по индексу, обеспечивают достаточно быстрый и эффективный доступ к данным в столбцах, не входящих в первичный ключ.

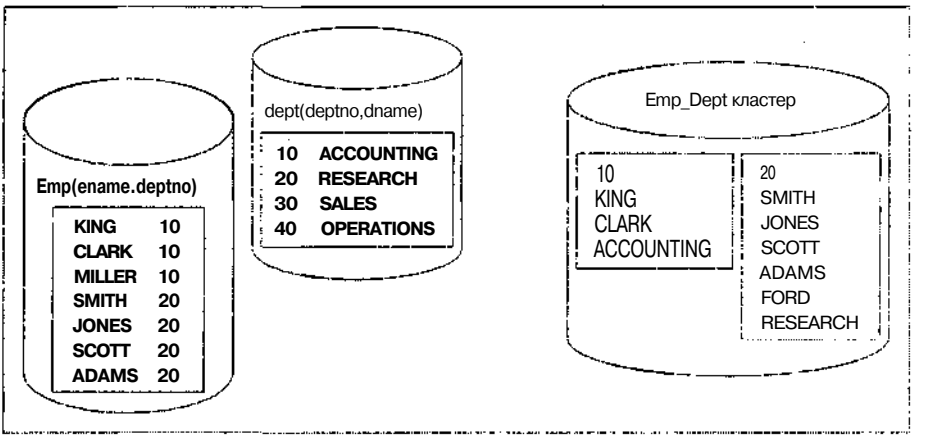
Итак, при использовании таблиц, организованных по индексу, важнее всего выбрать правильное сочетание данных, хранящихся в блоках индекса, и данных, хранящихся в сегменте остатка. Проверьте различные варианты с разными условиями попадания столбцов в сегмент остатка. Определите, как они влияют на выполнение операторов INSERT, UPDATE, DELETE и SELECT. Если таблица заполняется данными один раз и часто читается, поместите в блок индекса как можно больше данных. Если таблица часто изменяется, решите, что для вас лучше: поместить все данные в блок индекса (что хорошо для чтения) или часто реорганизовывать данные индекса (что плохо для изменений). То, что было сказано относительно списков FREELIST для обычных таблиц, относится и к таблицам, организованным по индексу. Параметры PCTFREE и PCTUSED для таблиц, организованных по индексу, имеют два назначения. Для таблицы, организованной по индексу, параметр PCTFREE не имеет такого значения, как для обычной таблицы, а параметр PCTUSED для нее вообще не используется. При создании сегмента OVERFLOW, однако, параметры PCTFREE и PCTUSED интерпретируются точно так же, как и для таблицы, организованной в виде кучи; их надо устанавливать для дополнительного сегмента, исходя из тех же соображений, что и для обычной таблицы.

Таблицы в индексном кластере

Я часто сталкиваюсь с тем, что кластеры в СУБД Oracle понимают неправильно. Многие путают их с "кластерным индексом" СУБД SQL Server или Sybase. Кластеры не имеют с ним ничего общего. *Кластер* — это способ хранения группы таблиц, имеющих один или несколько общих столбцов, в одних и тех же блоках базы данных, так что взаимосвязанные данные хранятся в одном блоке. Кластерный индекс в СУБД SQL Server позволяет хранить отсортированные по ключу индекса строки; он аналогичен описанной выше организации таблицы по индексу в Oracle. При использовании кластера блок данных может содержать данные из нескольких таблиц. По сути данные хранятся "предварительно соединенными". В кластер можно помещать и одну таблицу. При этом хранятся сгруппированные по значению некоторого столбца данные. Например, информация обо всех сотрудниках отдела 10 будет храниться в одном блоке (или в минимально возможном количестве блоков, если все записи в один блок не помещаются). Речь не идет о хранении отсортированных данных — для этого применяется таблица, организованная по

индексу. Это хранение кластеризованных по некоторому ключу данных, но в виде кучи. Поэтому строки для сотрудников отдела 100 могут находиться рядом со строками для отдела 1 и очень далеко (физически на диске) от строк для отделов 101 и 99.

Графически можно представить кластер так, как показано ниже. Слева мы используем обычные таблицы. Таблица **EMP** хранится в своем сегменте, а таблица **DEPT** — в своем. Они могут оказаться в разных файлах, в разных табличных пространствах и уж точно — в отдельных экстентах. Справа показано, что произойдет при кластеризации этих двух таблиц. Квадраты представляют блоки базы данных. Значение 10 теперь факторизовано и хранится один раз. Все данные из всех таблиц кластера для отдела 10 хранятся в том же блоке. Если все данные для отдела 10 не помещаются в один блок, к исходному блоку будут присоединены дополнительные блоки для размещения остатка, во многом аналогично дополнительным блокам таблиц, организованных по индексу:



Давайте разберемся, как создать кластеризованный объект. Создать кластер таблиц просто. Параметры хранения **(PCTFREE, PCTUSED, INITIAL** и т.д.) задаются для кластера, а не для таблиц. В этом есть смысл, поскольку в кластере будет много таблиц, и их строки будут в одном блоке. Разные значения параметра **PCTFREE** не имеют смысла. Поэтому оператор **CREATE CLUSTER** аналогичен оператору **CREATE TABLE** с небольшим количеством столбцов (указываются только столбцы ключа кластера):

```
tkyte@TKYTE816> create cluster emp_dept_cluster
  2   (deptno number(2))
  3   size 1024
  4   /
```

Cluster created.

Здесь мы создали индексный кластер (есть еще хеш-кластеры; мы рассмотрим их далее). Кластеризующим столбцом для этого кластера будет столбец **DEPTNO.** Столбцы в таблицах не обязательно должны называться **DEPTNO,** но они **должны** быть типа **NUMBER(2),** чтобы соответствовать определению. В определении кластера я указал опцию **SIZE 1024.** Она сообщает серверу Oracle, что с каждым значением кластерного клю-

ча предположительно будет связано 1024 байт данных. Сервер Oracle будет использовать это для вычисления **максимального** количества кластерных ключей, которые могут поместиться в блок. При использовании блоков размером 8 Кбайт сервер Oracle будет помещать в блок до семи ключей кластера (но, может, и меньше, если данных окажется больше, чем ожидалось). Другими словами, данные для отделов 10, 20, 30, 40, 50, 60, 70 скорее всего попадут в один блок, а при вставке данных для отдела 80 будет использован новый блок. Это не означает, что хранятся отсортированные данные — просто, если данные об отделах будут вставляться в таком порядке, они, естественно, будут храниться вместе. Если вставлять данные по отделам в следующем порядке: 10, 80, 20, 30, 40, 50, 60, а затем — 70, то последний отдел, 70, окажется в новом блоке. Как будет показано ниже, на количество хранящихся в блоке ключей влияет как размер данных, так и порядок их вставки.

Параметр SIZE, таким образом, контролирует максимальное количество ключей кластера в блоке. Он оказывает максимальное влияние на использование пространства в кластере. При установке слишком большого значения в блоке окажется очень мало ключей, и будет использоваться больше пространства, чем необходимо. Если значение слишком маленькое, данные будут слишком сильно фрагментированы, что противоречит цели создания кластера — хранить все данные вместе, в одном блоке. Это — один из важней^ших параметров для кластера.

Теперь перейдем к созданию индекса кластера. Индекс кластера должен быть создан до того, как в него начнут поступать данные. Можно создавать таблицы в кластере прямо сейчас, но я собираюсь одновременно создавать таблицы и наполнять их данными, а индекс кластера должен быть создан до вставки в него каких-либо данных. Задача индекса кластера — брать значение ключа кластера и возвращать адрес блока, содержащего это значение. Фактически это первичный ключ, в котором каждое значение ключа кластера указывает на один блок в самом кластере. Так что при запросе данных для отдела 10 сервер Oracle прочитает ключ кластера, определит адрес блока для этого ключа, а затем будет читать данные. Индекс по ключу кластера создается следующим образом:

```
tkyte@TKYTE816> create  index  emp_dept_cluster_idx
  2   on cluster emp_dept_cluster
  3  /

Index created.
```

При этом можно задавать все обычные параметры хранения индекса и помещать его в другое табличное пространство. Это самый обычный индекс, который просто индексирует ключи кластера и может также иметь отдельную запись для пустого значения (почему это существенно, см. в главе 7, посвященной индексам). Теперь все готово для создания таблиц в кластере:

```
tkyte@TKYTE816> create table dept
  2   (deptno number(2)  primary key,
  3    dname  varchar2 (14) ,
  4    loc    varchar2(13)
  5  )
  6  cluster einp_dept_cluster (deptno)
  7  /
```

```
Table created.

tkyte@TKYTE816> create table emp
  2   (empno number primary key,
  3    ename varchar2(10) ,
  4    job      varchar2(9),
  5    mgr      number,
  6    hiredate date,
  7    sal      number,
  8    comm     number,
  9    deptno number(2) references dept(deptno)
 10  )
 11   cluster emp_dept_cluster(deptno)
 12  /

Table created.
```

Здесь единственное отличие от "обычной" таблицы связано с использованием конструкции **CLUSTER,** которая сообщает серверу Oracle, какой столбец таблицы будет соответствовать ключу кластера. Теперь можно загружать в таблицу начальный набор данных:

```
tkyte@TKYTE816> begin
  2            for x in (select * from scott.dept)
  3            loop
  4                      insert into dept
  5                      values (x.deptno, x.dname, x.loc);
  6                      insert into emp
  7                      select *
  8                         from scott.emp
  9                        where deptno = x.deptno;
 10            end loop;
 11  end;
 12  /

PL/SQL procedure successfully completed.
```

Возможно, вы задаете себе вопрос: "Почему бы просто не вставить все данные таблицы **DEPT,** а потом все данные таблицы **EMP;** или почему мы загружаем данные вот так, по отделам?". Причина в структуре кластера. Я имитировал начальную загрузку большого объема данных в кластер. Если бы я сначала загрузил все строки из таблицы **DEPT,** мы бы точно получили по 7 ключей в блоке (в соответствии с заданным параметром **SIZE 1024),** поскольку строки таблицы **DEPT** — очень маленькие, всего несколько байтов. Когда же дело дойдет до загрузки строк таблицы **EMP,** может оказаться, что в некоторых отделах данных намного больше, чем 1024 байт. В результате потребуется создание больших цепочек блоков для соответствующих ключей кластера. Загружая все данные для одного ключа кластера сразу, мы максимально упаковываем блоки и начинаем новый блок, когда в текущем уже нет места. Сервер Oracle разместит не до семи ключей кластера в одном блоке, а столько, сколько вместится. Простой пример покажет различие между двумя подходами к загрузке данных.

Добавим в таблицу EMP большой столбец типа **CHAR(1000)**. Он сделает строки таблицы EMP намного больше. Будем загружать таблицы кластера двумя способами. Сначала загрузим всю таблицу DEPT, а затем — всю таблицу EMP. После этого будем загружать данные по отделам: строку — из таблицы DEPT, а затем — все строки из таблицы EMP с соответствующим номером отдела; потом — следующую строку из таблицы DEPT. Посмотрим, в каком блоке окажутся строки в каждом из случаев, чтобы понять, какой из способов больше соответствует цели совместного размещения данных с общим значением в столбце DEPTNO. В этом примере таблица EMP имеет следующий вид:

```
create table emp
(empno number primary key,
 ename varchar2(10),
 job    varchar2(9),
 mgr    number,
 hiredate date,
 sal    number,
 conm   number,
 deptno number(2) references dept(deptno) ,
 data   char(1000) default '*'
)
cluster emp_dept_cluster(deptno)
/
```

При загрузке данных последовательно в таблицы DEPT и EMP окажется, что многие строки таблицы EMP больше не попадают в тот же блок, что и строка DEPT (DBMS_ROWID — это стандартный пакет, позволяющий анализировать значение идентификатора строки):

```
tkyte@TKYTE816> insert into dept
  2   select * from scott.dept
  3   /

4 rows created.
tkyte@TKYTE816> insert into emp
  2   select emp.*, '*' from scott.emp
  3   /

14 rows created.

tkyte@TKYTE816> select dbms_rowid.rowid_block_number(dept.rowid) dept_rid,
  2           dbms_rowid.rowid_block_number(emp.rowid) emp_rid,
  3                dept.deptno
  4    from emp, dept
  5    where emp.deptno = dept.deptno
  6   /

  DEPT_RID   EMP_RID   DEPTNO

        10        12       10
        10        11       10
        10        11       10
        10        10       20
        10        10       20
```

```
        10         12        20
        10         11        20
        10         11        20
        10         10        30
        10         10        30
        10         10        30
        10         10        30
        10         11        30
        10        11         30

14 rows selected.
```

Более половины строк таблицы EMP не попали в блок с соответствующей строкой таблицы DEPT. При загрузке данных не по ключам таблиц, а по ключу кластера получаем:

```
tkyte@TKYTE816> begin
  2       for x in  (select  *  from scott.dept)
  3       loop
  4            insert  into dept
  5            values  (x.deptno, x.dname, x.loc);
  6            insert  into emp
  7            select emp.*,  'x'
  8               from scott.emp
  9              where  deptno  =  x.deptno;
 10       end loop;
 11   end;
 12   /

PL/SQL procedure  successfully completed.

tkyte@TKYTE816> select dbms_rowid.rowid_block_number(dept.rowid)  dept_rid,
  2      dbms_rowid.rowid_block_number      (emp.rowid)  emp_rid,
  3                 dept.deptno
  4    from emp,  dept
  5    where  emp.deptno  =  dept.deptno
  6   /

  DEPT_RID    EMP_RID    DEPTNO

        11          11        30
        11          11        30
        11          11        30
        11          11        30
        11          11        30
        11          11        30
        12          12        10
        12          12        10
        12          12        10
        12          12        20
        12          12        20
        12          12        20
```

```
14 rows selected.
```

Большинство строк таблицы EMP теперь в том же блоке, что и соответствующая строка таблицы DEPT. Этот пример — несколько надуманный, поскольку я умышленно занизил параметр SIZE для кластера, чтобы продемонстрировать проблему более наглядно, но рекомендуемый метод подходит для начальной загрузки кластера. Он гарантирует, что если для некоторых ключей кластера предполагаемое значение SIZE будет превышено, большинство данных все равно будет кластеризовано в одном блоке. Если загружать таблицы поочередно, так не получится.

Это применимо только для начальной загрузки кластера. После этого кластер используется так, как необходимо транзакциям, — адаптировать приложение специально для работы с кластером не требуется.

Вот вам небольшая головоломка, чтобы удивить друзей. Многие ошибочно полагают, что идентификатор строки уникально идентифицирует ее в базе данных, т.е. что по идентификатору строки можно определить, из какой она таблицы. Не самом деле идентификаторы строк таблиц в кластере дублируются. Например, после выполнения предыдущих действий получается:

```
tkyte@TKYTE816> select rowid from emp
  2  intersect
  3  select rowid from dept;

ROWID
------------------
AAAGBOAAFAAAAJyAAA
AAAGBOAAFAAAAJyAAB
AAAGBOAAFAAAAJyAAC
AAAGBOAAFAAAAJyAAD
```

Идентификаторы строк в таблице DEPT выделялись и строкам в таблице EMP. Дело в том, что для уникальной идентификации строки надо знать таблицу и идентификатор строки. Псевдостолбец rowid уникален только в пределах таблицы.

Я также обнаружил, что многие считают кластер объектом загадочным и реально не используемым. Все используют обычные таблицы. Фактически же кластеры используются при любом обращении к СУБД Oracle. Значительная часть словаря данных хранится в различных кластерах. Например:

```
sys@TKYTE816> select cluster_name, tablename from user_tables
  2  where cluster_name is not null
  3  order by 1
  4  /

CLUSTER NAME          TABLE KAME
--------------------- ----------------------
C_COBJ#               CCOL$
                      CDEF$
C FILE# BLOCK#        SEG$
```

```
                    UET$
C_MLOG#             MLOG$
                    SLOG$
C_OBJ#              ATTRCOL$
                    COL$
                    COLTYPE$
                    CLUS
                    ICOLDEP$
                    LIBRARY$
                    LOB$
                    VIEWTRCOL$
                    TYPE_MISC$
                    TAB$
                    REFCON$
                    NTAB$
                    IND$
                    ICOL$
C_OBJ#_INTCOL#      HISTGRM$
C_RG#               RGCHILD$
                    RGROUPS
C_TOID_VERSION#     ATTRIBUTE$
                    COLLECTION$
                    METHOD$
                    RESULT$
                    TYPE$
                    PARAMETER$
C_TS#               FET$
                    TS$
C_OSER#             TSQ$
                    USER$

33 rows selected.
```

Как видите, большинство связанных с объектами данных хранится в одном кластере (в кластере **C_OBJ#),** 14 таблиц используют одни и те же блоки. В нем хранится в основном информация о столбцах, поэтому вся информация о наборе столбцов таблицы или индекса хранится физически в том же блоке. В этом есть смысл; когда сервер Oracle анализирует запрос, ему необходим доступ к данным обо всех столбцах соответствующей таблицы. Если эти данные разбросаны, для их сбора потребуется больше времени. Здесь же они обычно находятся в одном блоке и сразу доступны.

Когда же использовать кластеры? Пожалуй, проще сформулировать, когда их не надо использовать.

- **Кластеры могут отрицательно сказаться на производительности операторов ЯМД.** Если предполагается активное изменение таблиц в кластере, необходимо учитывать, что индексный кластер будет снижать производительность. Для управления данными в кластере необходимо выполнить больше действий.
- **Полный просмотр таблиц в кластере выполняется медленнее.** Приходится просмотреть больше данных, поскольку вместо данных только одной таблицы просмат-

риваются данные нескольких таблиц. Поэтому полный просмотр выполняется дольше.

- **Если предполагается частое усечение (TRUNCATE) и загрузка данных в таблицу.** Таблицы в кластере нельзя усечь. Это очевидно, поскольку в блоках кластера хранятся данные нескольких таблиц, строки в таблице, входящей в кластер, придется просто удалять.

Итак, если данные в основном считываются (это **не** означает "никогда не изменяются" — данные таблиц в кластере можно изменять), причем — по индексам (либо по индексу кластера, либо по другим индексам) и полученная информация часто соединяется с другими таблицами кластера, кластер имеет смысл использовать. Ищите таблицы, логически взаимосвязанные и используемые совместно, по примеру разработчиков словаря данных Oracle, кластеризовавших вместе всю информацию о столбцах.

Итак, размещение таблиц в кластере позволяет заранее соединить их данные. Кластеры используются для хранения взаимосвязанных данных из многих таблиц в одном блоке базы данных. Кластеры позволяют ускорить операции, с помощью которых интенсивно считываются данные и соединяются таблицы или происходит обращение к взаимосвязанным данным (например, для получения информации обо всех сотрудниках отдела 10). Они сокращают количество блоков, которые должен кэшировать сервер Oracle; вместо хранения 10 блоков для 10 сотрудников одного отдела, будет храниться один блок, т.е. увеличится эффективность буферного кэша. С другой стороны, если неправильно задать параметр **SIZE,** кластеры могут неэффективно использовать пространство и замедлять работу операторов ЯМД.

Таблицы в хеш-кластере

Таблицы в хеш-кластере по сути очень похожи на представленные выше таблицы в индексном кластере, за одним исключением: вместо индекса по ключу кластера используется хеш-функция. Данные в таблице и есть индекс — отдельного индекса нет. Сервер Oracle берет значение ключа для строки, хеширует его с помощью внутренней или указанной администратором функции и использует результат для определения местонахождения данных на диске. Однако при использовании для поиска данных алгоритма хеширования невозможно просматривать диапазон значений ключей в хеш-кластере, не добавив для соответствующей таблицы обычный индекс. В рассмотренном ранее индексном кластере при выполнении запроса:

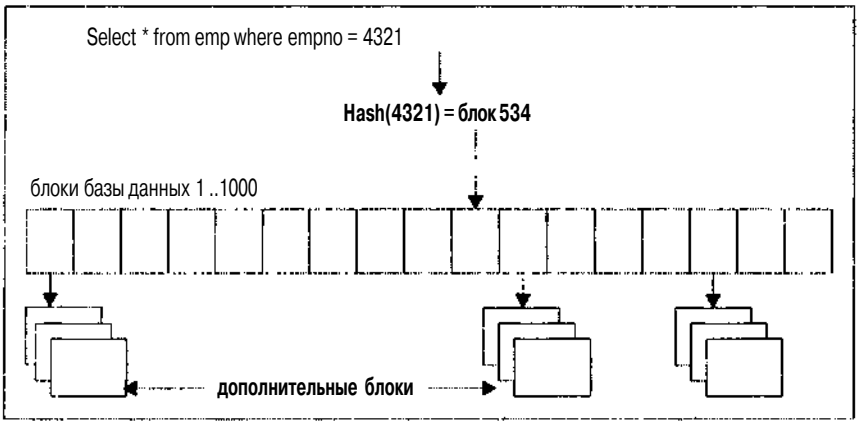
```
select * from emp where deptno between 10 and 20
```

сервер сможет использовать для поиска строк индекс по ключу кластера. В хеш-кластере этот запрос будет выполняться путем полного просмотра таблицы, если нет индекса по столбцу **DEPTNO.** По хеш-ключу без индекса, поддерживающего просмотр диапазонов, можно выполнять только поиск по равенству.

В идеальном случае, когда алгоритм хеширования дает минимум совпадений, хеш-кластер позволял бы получить запрашиваемые данные с помощью одной операции ввода/вывода. В действительности же хеш-значения иногда совпадают, кроме того, возможна фрагментация строк, так что для извлечения некоторых данных потребуется больше одной операции ввода/вывода.

Как и хеш-таблицы в языках программирования, хеш-таблицы в базе данных имеют фиксированный размер. При создании таблицы необходимо раз и навсегда определить, сколько хеш-ключей будет иметь таблица. Это не ограничивает количество строк, которые можно в нее поместить.

Ниже графически представлен хеш-кластер, в котором создана таблица **EMP.** Когда клиент посылает запрос, использующий в условии ключ хеш-кластера, сервер Oracle применяет хеш-функцию, чтобы определить, в каком блоке искать данные. Если хеш-значения многократно повторяются или параметр **SIZE** в операторе **CREATE CLUSTER** имел слишком маленькое значение, серверу Oracle приходится выделять блоки остатка, связанные в цепочку с исходным блоком.



При создании хеш-кластера используется тот же оператор **CREATE CLUSTER,** что и при использовании индексного кластера, но с другими опциями. Достаточно просто добавить опцию **HASHKEYS,** задающую размер хеш-таблицы. Сервер Oracle округляет значение **HASHKEYS** до ближайшего простого числа (поэтому количество хеш-ключей всегда выражено простым числом). Затем сервер Oracle умножает значение параметра SIZE на измененное значение **HASHKEYS.** После этого на диске выделяется под кластер соответствующее количество байтов. Это существенно отличается от индексного кластера, где пространство выделяется динамически, по мере надобности. Хеш-кластер же заранее выделяет пространство, достаточное для размещения **(HASHKEYS/ trunc(blocksize/SIZE)>** байтов данных. Так что, например, если значение **SIZE** установлено равным 1500 байт и используются блоки размером 4 Кбайт, сервер Oracle будет планировать хранение в блоке двух ключей. Если предполагается наличие 1000 хеш-ключей, сервер Oracle выделит под кластер 500 блоков.

Интересно отметить, что, в отличие от обычной хеш-таблицы в языках программирования, совпадение хеш-значений вполне допустимо, а во многих случаях даже желательно. Вернувшись к примеру с таблицами **DEPT/EMP,** можно создать хеш-кластер по столбцу **DEPTNO.** Очевидно, что многие строки будут иметь одинаковое хеш-значение; это и предполагается, поскольку они имеют одинаковое значение **DEPTNO.** Кластер в общем-то и создается для совместного размещения сходных данных. Вот почему сервер Oracle требует указать значение параметров **HASHKEYS** (какое количество от-

делов предполагается) и SIZE (какой объем данных будет ассоциироваться с каждым номером отдела). Он создает хеш-таблицу для хранения указанного количества отделов размером SIZE байтов каждый. Избегать надо непреднамеренных совпадений хеш-значений. Очевидно, что, если установить размер хеш-таблицы 1000 (фактически таблица будет иметь размер 1009, поскольку размер хеш-таблицы всегда выражается простым числом и сервер Oracle автоматически округляет переданное значение) и поместить в таблицу информацию о 1010 отделах, хотя бы одно совпадение точно будет (два разных номера отдела будут хешированы в одно и то же значение). Непреднамеренных совпадений хеш-значений следует избегать, поскольку они увеличивают расход ресурсов и вероятность фрагментации строк.

Чтобы разобраться, как используется пространство для хеш-кластера, напишем небольшую служебную хранимую процедуру SHOW_SPACE, которая будет использоваться в этой и в следующей главе, посвященной индексам. Эта процедура использует подпрограммы пакета DBMS_SPACE, частично уже продемонстрированные ранее, для выдачи информации о занимаемом объектами базы данных пространстве:

```
tkyte@TKYTE816> create or replace
  2  procedure show_space
  3  (p_segname  in varchar2,
  4   p_owner    in varchar2 default user,
  5   p_type     in varchar2 default  'TABLE',
  6   p_partition in varchar2 default NULL)
  7  as
  8        l_free_blks           number;
  9
 10        l_total blocks        number;
 11        l_total_bytes         number;
 12        l_unused_bloclcs      number;
 13        l_unused_bytes        number;
 14        l_LastUsedExtFileId   number;
 15     l_LastOsedExtBlockId     number;
 16        l_last_used_block     number;
 17       procedure p(p_label  in varchar2,  p_num in number)
 18        is
 19        begin
 20             dbms_output.put_line(rpad(p_label,40,'.') ||
 21                                       p_num);
 22        end;
 23  begin
 24        dbms_space.free_blocks
 25        (segment_owner        => p_owner,
 26         segment_name         => p_segname,
 27         segment_type         => p_type,
 28              partition_name  => p_partition,
 29         freelist_group_id    => 0,
 30         free_blks            -> l_free_blks);
 31
 32        dbms_space.unused_space
 33        (segment_ovmer        => p_owner,
```

```
 34         segment_name          => p_segname,
 35         segment_type          => p_type,
 36              partition_name   => p_partition,
 37         total_blocks          => l_total_blocks,
 38         total_bytes           => l_total_bytes,
 39         unused_blocks         => l_unused_blocks,
 40         unused_bytes          => l_unused_bytes,
 41         last_used_extent_file_id => l_LastUsedExtFileId,
 42         last_used_extent_block_id => l_LastUsedExtBlockId,
 43        last_used_block => l_last_used_block);
 44
 45       p('Free Blocks', l_free_blks);
 46       p('Total Blocks', l_total_blocke);
 47       p('Total Bytes', l_total_bytes);
 48       p('Unused Blocks', l_unused_blocks);
 49       p('Unused Bytes', l_unused_bytes);
 50    p('Last    Used Ext Fileld', l_LastUsedExtFilerd) ;
 51    p('Last    Used Ext Blockld', l_LastUsedExtBlockId) ;
 52       p('Last Used Block', l_last_used_block);
 53  end;
 54  /

Procedure created.
```

Теперь, выполнив оператор CREATE CLUSTER, можно увидеть, сколько места выделено под кластер:

```
tkyte@TKYTE816> create cluster hash_cluster
  2   (hash_key number)
  3   hashkeys 1000
  4   size 8192
  5   /

Cluster created.

tkyte@TKYTE816> exec show_space('HASH_CLUSTER', user, 'CLUSTER')
Free Blocks.....................0
Total Blocks....................1016
Total Bytes.....................8323072
Unused Blocks...................6
Unused Bytes....................49152
Last Used Ext Fileld............5
Last Used Ext Blockld...........889
Last Used Block.................2

PL/SQL procedure successfully completed.
```

Итак, всего кластеру выделено 1016 блоков. Шесть из этих блоков не используются (свободны). Один блок используется под служебную информацию объекта для управления экстентами. Поэтому до отметки максимального уровня в этом объекте имеется 1009 блоков, и все они используются кластером. 1009 — наименьшее простое число, большее

1000, а поскольку у меня размер блока — 8 Кбайт, то сервер Oracle должен был выделить (8192 * 1009) байт. Мы получили несколько больший результат; это связано с округлением количества выделяемых экстентов и/или использованием локально управляемого табличного пространства с экстентами одинакового размера.

Мы столкнулись с одной из проблем при использовании хеш-кластеров, которую необходимо учитывать. Обычно при создании пустой таблицы количество блоков до отметки максимального уровня равно 0. При ее полном просмотре сразу достигается отметка максимального уровня, и просмотр завершается. При использовании хеш-кластера таблицы — изначально большие и создаваться будут дольше, поскольку сервер Oracle должен проинициализировать каждый блок, что обычно происходит лишь при добавлении данных в таблицу. Таблицы могут иметь данные в первом и в последнем блоке, а остальные будут пустыми. Полный просмотр практически пустого хеш-кластера потребует столько же времени, как и полного. Это не всегда плохо: хеш-кластер создается для очень быстрого доступа к данным по ключу, а не для частого полного просмотра.

Теперь можно помещать таблицы в хеш-кластер точно так же, как это делается для индексных кластеров. Например:

```
tkyte@TKYTE816> create table hashed_table
  2   (x number, data1 varchar2(4000), data2 varchar2(4000))
  3   cluster hash_cluster(x);

Table created.
```

Чтобы продемонстрировать отличие хеш-кластера, я создал небольшой тестовый пример: создал хеш-кластер, загрузил в него немного данных, скопировал эти данные в "обычную" таблицу с обычным индексом, а затем выполнил 100000 случайных чтений из каждой таблицы (в равной степени случайные чтения). С помощью параметра SQL_TRACE и утилиты TKPROF (подробнее об этих средствах см. в главе 10, посвященной стратегиям и средствам настройки) я определил производительность работы с каждой из них. Вот что я делал и как анализировал результаты:

```
tkyte@TKYTE816> create cluster hash_cluster
  2    (hash_key number)
  3   hashkeys 50000
  4   size 45
  5   /

Cluster created.

tkyte@TKYTE816> create table emp
  2   cluster hash_cluster(empno)
  3   as
  4   select rownum empno, ename, job, mgr, hiredate, sal, comra, deptno
  5      from scott.emp
  6    where 1=0
  7   /

Table created.
```

Определив, что средний размер строки в таблице будет 45 байт (для этого я проанализировал таблицу SCOTT.EMP), я создал хеш-кластер с параметром SIZE, равным 45

байт. Затем я создал в кластере пустую таблицу, по структуре аналогичную таблице **SCOTT.EMP.** Единственное изменение — выбор **ROWNUM** вместо **EMPNO,** так что в созданной мной таблице этот столбец имеет тип NUMBER, а не NUMBER(4). В таблице заведомо должно быть более 9999 строк, поскольку я собираюсь вставить туда около 50000.

Затем я заполнил данными таблицу и создал ее "обычный" аналог:

```
tkyte@TKYTE816> declare
  2          1 cnt    number;
  3          l_empno number default 1;
  4   begin
  5          select count(*)  into l_cnt from scott.emp;
  6
  7          for x in  (select * from scott.emp)
  8          loop
  9             for i in 1  ..  trunc(50000/l_cnt)+1
 10             loop
 11                   insert into emp values
 12                  (l_empno,  x.ename,  x.job,  x.mgr,  x.hiredate,  x.sal,
 13            x.comm,            x.deptno) ;
 14                   l_empno  := l_empno+1;
 15             end loop;
 16          end loop;
 17          commit;
 18   end;
 19   /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> create table emp_reg
  2   as
  3   select * from emp;

Table created.

tkyte@TKYTE816> alter table emp_reg add constraint emp_pk primary
key(empno);

Table altered.
```

Теперь осталось только получить "случайные" данные для выборки строк из каждой таблицы:

```
tkyte@TKYTE816> create table random  (x int) ;

Table created.

tkyte@TKYTE816> begin
  2            for i in 1  . .  100000
  3             loop
  4                    insert into random values
  5                    (mod (abs(dbms_random.random),50000)+1);
  6             end loop;
```

```
  7  end;
  8  /

PL/SQL procedure successfully completed.
```

Теперь все готово для тестирования:

```
tkyte@TKYTE816> alter session set sql_trace=true;

Session altered.

tkyte@TKYTE816> select count(ename)
  2     from emp, random
  3    where emp.empno = random.x;

 COUNT(ENAME)
-------------


tkyte@TKYTE816> select count(ename)
  2     from emp_reg, random
  3    where emp_reg.empno = random.x;

 COUNT(ENAME)
-------------
       100000
```

Я знал, что оптимизатор должен выбрать метод FULL SCAN random в обоих случаях, поскольку другого метода доступа к этой таблице не существует. Я полагал, что будет выполняться соединение вложенными циклами с таблицами EMP и EMP_REG (что и происходило). В результате было выполнено 100000 случайных чтений из двух этих таблиц. В отчете TKPROF я обнаружил следующее:

```
select count(ename)
  from emp, random
 where emp.empno = random.x

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          2           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      3.44       3.57         13     177348          4           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      3.44       3.57         13     177348          6           1

Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 66

Rows     Row Source Operation

      1  SORT AGGREGATE
 100000   NESTED LOOPS
 100001    TABLE ACCESS FULL RANDOM
 100000    TABLE ACCESS HASH EHF
********************************************************************************
```

```
select count(ename)
  from emp_reg, random
 where emp_reg.empno = random.x

call      count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ---------  ----------  ----------  ----------  ----------
Paraa         1      0.01       0.01          0           1           3           0
Execute       1      0.00       0.00          0           0           0           0
Fetch         2      1.80       6.26        410      300153           4           1
-------  ------  --------  ---------  ----------  ----------  ----------  ----------
total         4      1.81       6.27        410      300154           7           1

Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 66

Rows     Row Source Operation

      1  SORT AGGREGATE
 100000   NESTED LOOPS
 100001    TABLE ACCESS FULL RANDOM
 100000    TABLE ACCESS BY INDEX ROWID EMP_REG
 200000     INDEX UNIQUE SCAN (object id 24743)
```

Хочу обратить ваше внимание на следующее.

- Хеш-кластер требует существенно меньше операций ввода/вывода (столбец query). Этого мы и ждали. При выполнении запроса брались случайные числа, к ним применялась хеш-функция и читался соответствующий блок. Для получения данных из хеш-кластера необходима минимум одна операция ввода/вывода. В обычной таблице с индексом придется просмотреть индекс, а затем выполнить доступ к таблице по идентификатору строки. Для получения данных из таблицы по индексу необходимо минимум две операции ввода/вывода.
- Запрос к хеш-кластеру требует заметно больше процессорного времени. Это тоже можно было предсказать. Вычисление хеш-значения требует ресурсов процессора. Поиск по индексу требует выполнения нескольких операций ввода/вывода.
- Общее время выполнения запроса к хеш-кластеру — меньше. Так будет не всегда. В моей системе (для этого теста использовался ноутбук с одним пользователем; медленно работающие диски, но все процессорное время принадлежит мне), ресурсов процессора хватало, но медленно работал диск. Поскольку я имел исключительный доступ к ресурсам процессора, время выполнения для запроса к хеш-кластеру оказалось близким к процессорному времени выполнения. Однако, поскольку диски моего ноутбука работают не слишком быстро, пришлось долго ждать выполнения операций ввода/вывода.

Последнее замечание наиболее существенно. При работе с компьютерами необходимо учитывать ресурсы и их использование. Если низка производительность процесса ввода/вывода и выполняются запросы, интенсивно выбирающие данные по ключу, как в приведенном примере, хеш-кластер может повысить производительность. Если же не

хватает ресурсов процессора, хеш-кластер только снизит производительность, поскольку требует дополнительных вычислений. Это одна из основных причин, почему простые правила не работают в реальных ситуациях: что сработало один раз, может не произойти в похожей ситуации, но при других условиях.

Особым случаем хеш-кластера является однотабличный хеш-кластер. Это — оптимизированная версия уже рассмотренного хеш-кластера общего вида. В таком кластере в каждый момент времени может находиться только одна таблица (необходимо удалить существующую таблицу в однотабличном хеш-кластере, прежде чем можно будет создать новую). Кроме того, при наличии однозначного соответствия между хеш-значениями и строкам данных, доступ к данным несколько ускоряется. Эти хеш-кластеры создавались для случаев, когда необходим быстрый доступ к таблице по первичному ключу, а не совместное размещение нескольких таблиц. Если необходим быстрый доступ к записи сотрудника по полю **EMPNO,** имеет смысл подумать об использовании однотабличного хеш-кластера. Я выполнил тест, аналогичный описанному выше, на однотабличном хеш-кластере, и получил еще более высокую производительность, чем при использовании обычного хеш-кластера. Однако в этом примере я пошел дальше, т.е. учел тот факт, что сервер Oracle позволяет задать собственную, специализированную хеш-функцию (вместо стандартной функции сервера). В хеш-функциях можно использовать только столбцы таблицы и встроенные функции Oracle (нельзя использовать свои хранимые функции, написанные, например, на языке PL/SQL). Учитывая, что столбец EMPNO в предыдущем примере — число от 1 до 50000, я создал свою "хеш-функцию" в виде столбца **EMPNO.** При этом гарантировано не будет совпадения хеш-значений. Итак, создаем однотабличный хеш-кластер с собственной хеш-функцией:

```
tkyte@TKYTE816>  create  cluster  single_table_hash_cluster
  2   (hash_key   INT)
  3   hashkeys  50000
  4   size  45
  5   single  table
  6  hash is HASH_KEY
  7   /

Cluster  created.
```

Для того чтобы сделать хеш-кластер однотабличным, мы добавили ключевые слова **SINGLE TABLE.** Функция **HASH IS** в данном случае — просто ключ кластера, **HASH_KEY.** Это SQL-функция; при желании можно было бы использовать выражение **trunc(mod(hash_key/324+278555)/abs(hash_key+l))** (я не утверждаю, что это хорошая хеш-функция, просто демонстрирую, что при необходимости можно задавать сложное выражение). Теперь создадим таблицу в кластере:

```
tkyte@TKYTE816> create table single_table_emp
  2   (empno  INT  ,
  3    ename  varchar2(10) ,
  4    job   varchar2(9),
  5    mgr   number,
  6    hiredate      date,
```

```
  7    sal  number,
  8    comm number,
  9    deptno  number(2)
 10  )
 11  cluster single_table_hash_cluster(empno)
 12  /
```

Table created.

и загрузим в нее данные из таблицы **EMP,** оставшиеся в этой таблице после выполнения предыдущего примера:

```
tkyte@TKYTE816> insert into single_table_emp
  2   select * from emp;

50008 rows created.
```

После выполнения того же запроса, что и для прежних двух таблиц, в отчете утилиты **TKPROF** можно обнаружить следующее:

```
select count(ename)
  from single_table_emp, random
 where single_table_emp.empno = random.x
call       count      cpu    elapsed    disk    query    current     rows
-------------------------------------------------------------------------
Parse          1     0.00       0.00       0        0          0        0
Execute        1     0.00       0.00       0        0          0        0
Fetch          2     3.29       3.44     127   135406          4        1
-------------------------------------------------------------------------
total          4     3.29       3.44     127   135406          4        1

Misses in library cache during parse: 0
Optimizer goal: CHOOSE
Parsing user id: 264

Rows     Row Source Operation

     1   SORT AGGREGATE
100000    NESTED LOOPS
100001      TABLE ACCESS FULL RANDOM
100000      TABLE ACCESS HASH SINGLE_TABLE_EMP
```

Запрос обработал на четверть меньше блоков, чем в обычном хеш-кластере. Это произошло благодаря удачному сочетанию использования собственной хеш-функции, гарантирующей несовпадение ключей, и однотабличного хеш-кластера.

Мы рассмотрели основные особенности хеш-кластеров. Они сходны с рассмотренными ранее индексными кластерами, но при обращении к данным не используется индекс, поскольку сами данные и являются индексом. Ключ кластера хешируется в адрес блока, и предполагается, что данные находятся в этом блоке. Важно хорошо понять следующее.

- Место под хеш-кластер выделяется сразу при создании. Сервер Oracle выделит **HASHKEYS/trunc(blocksize/SIZE)** байтов сразу же. При помещении в кластер

хотя бы одной таблицы, при полном просмотре будут затрагиваться все выделенные блоки. Это отличает таблицы в хеш-кластере от всех остальных таблиц.

- Количество ключей хеш-кластера фиксировано. Нельзя изменить размер хеш-таблицы, не перестраивая кластер. Но это никак не ограничивает объем данных, которые можно хранить в кластере, — ограничивается только количество генерируемых для кластера хеш-ключей. Это может повлиять на производительность из-за незапланированного совпадения ключей при слишком малом значении параметра **HASHKEYS.**
- Эффективный просмотр диапазонов значений ключа кластера невозможен. При проверке условий типа **WHERE cluster_key BETWEEN 50 AND 60** не может использоваться алгоритм хеширования. Между значениями 50 и 60 много возможных значений, и серверу пришлось бы все их сгенерировать, чтобы хешировать каждое и найти соответствующие данные. Это невозможно. При поиске по диапазону значений ключа хеш-кластер будет просматриваться полностью, если нет отдельного обычного индекса.

Хеш-кластеры можно использовать при следующих условиях.

- Достаточно точно известно, сколько строк будет в таблице, или можно указать обоснованный верхний предел количества строк. Правильная установка значений параметров **HASHKEYS** и **SIZE** позволит избежать пересоздания. Если таблица существует недолго (например, в витринах или хранилищах данных), подобрать значения несложно.
- Операторы ЯМД, особенно вставки, выполняются легко. Изменения тоже не требуют дополнительных расходов ресурсов, если только не изменяется HASHKEY, чего лучше не делать, потому что придется переносить строку.
- Доступ к данным происходит по значению ключа кластера, **HASHKEY.** Например, имеется таблица запасных частей, и доступ к ней выполняется по коду запасной части. Таблицы-справочники особенно удобно размещать в хеш-кластерах.

Вложенные таблицы

Вложенные таблицы — часть объектно-реляционных расширений (Object Relational Extensions) СУБД Oracle. Вложенная таблица, один из двух типов наборов в Oracle, очень похожа на подчиненную таблицу в традиционной для реляционной модели паре таблиц главная/подчиненная. Это неупорядоченный набор элементов данных одного типа, встроенного или объектного. Но при использовании вложенных таблиц создается впечатление, что каждая строка в главной таблице имеет отдельную подчиненную таблицу. Если в главной таблице — 100 строк, то имеется 100 **виртуальных** вложенных таблиц. Физически же имеется только одна главная и одна подчиненная таблица. Кроме того, между вложенными и главными/подчиненными таблицами есть много синтаксических и семантических различий, которые мы рассмотрим в этом разделе.

Вложенные таблицы можно использовать двумя способами. Один, — в PL/SQL-коде как средство расширения языка PL/SQL. Этот способ мы рассмотрим в главе 20. Другой, — как физический механизм хранения данных, для постоянного хранения наборов. Лично я постоянно использую их при программировании на PL/SQL и очень редко для постоянного хранения данных.

В этом разделе я кратко представлю синтаксис операторов для создания вложенных таблиц, выполнения к ним запросов и их изменения. Затем мы разберемся с подробностями реализации: что необходимо знать о фактическом хранении вложенных таблиц в СУБД Oracle.

Синтаксис вложенных таблиц

Создать таблицу с вложенной таблицей достаточно просто, а вот синтаксис операторов для работы с ними — несколько сложнее. Для демонстрации я буду использовать простые таблицы EMP и DEPT. Мы уже знакомы с этой небольшой реляционной моделью данных, которая реализуется следующим образом:

```
tkyte@TKYTE816> create table dept
  2   (deptno number(2) primary key,
  3       dname      varchar2(14),
  4    loc           varchar2(13)
  5   );

Table created.

tkyte@TKYTE816> create table emp
  2   (empno         number(4) primary key,
  3    ename         varchar2(10),
  4    job           varchar2(9),
  5    mgr           number(4) references emp,
  6    hiredate      date,
  7    sal           number(7, 2),
  8    comm          number(7, 2),
  9    deptno        number(2) references dept
 10   );
```

Table created:

с помощью первичного и внешнего ключей. Мы реализуем эту же модель с помощью вложенной таблицы, содержащей данные о сотрудниках:

```
tkyte@TKYTE816> create or replace type emp_type
  2   as object
  3   (empno         number(4),
  4    ename         varchar2(10),
  5    job           varchar2(9),
  6    mgr           number(4),
  7    hiredate      date,
  8    sal           number(7, 2),
  9    comm          number(7, 2)
```

```
  10  );
  11  /

Type created.

tkyte@TKYTE816> create or replace type emp_tab_type
  2   as table of emp_type
  3   /

Type created.
```

Для создания таблицы с вложенной таблицей необходим тип данных для вложенной таблицы. Представленный выше код создает сложный объектный тип EMP_TYPE и тип вложенной таблицы, который называется EMP_TAB_TYPE. В языке PL/SQL с ним можно было бы работать как с массивом. В языке SQL будет создана физическая вложенная таблица. Вот простой оператор CREATE TABLE, использующий этот тип:

```
tkyte@TKYTE816> create table dept_and_emp
  2   (deptno number(2) primary key,
  3    dname      varchar2(14),
  4    loc         varchar2(13),
  5    amps        emp_tab_type
  6   )
  7   nested table emps store as emps_nt;

Table created.

tkyte@TKYTE816> alter table emps_nt add constraint emps_empno_unique
  2                unique(empno)
  3   /

Table altered.
```

Важная часть этого оператора создания таблицы — включение столбца EMPS типа EMP_TAB_TYPE и соответствующая конструкция NESTED TABLE EMPS STORE AS EMPS_NT. При этом, помимо таблицы DEPT_AND_EMP, отдельно создается реальная физическая таблица EMPS_NT. Я добавил ограничение по столбцу EMPNO непосредственно для вложенной таблицы, чтобы обеспечить уникальность значения EMPN0, как это было в исходной реляционной модели. Я не могу реализовать всю модель данных. Попытаюсь добавить требование ссылки на саму себя:

```
tkyte@TKYTE816> alter table emps_nt add constraint mgr_fk
  2   foreign key(mgr) references emps_nt(empno);
alter table emps_nt add constraint mgr_fk
*

ERROR at line 1:
ORA-30730: referential constraint not allowed on nested table column
```

Однако это не срабатывает. Вложенные таблицы не поддерживают требования целостности ссылок, поскольку не могут ссылаться на другие таблицы, даже на самих себя. Поэтому пока оставим все как есть. Теперь давайте заполним таблицу данными из существующих таблиц EMP и DEPT:

```
tkyte@TKYTE816> insert into dept_and_emp
  2  select dept.*,
  3     CAST(multiset(select empno, ename, job, mgr, hiredate, sal, comm
  4                     from SCOTT.EMP
  5                    where emp.deptno = dept.deptno)  AS emp_tab_type)
  6     from SCOTT.DEPT
  7  /

4 rows created.
```

Здесь хочу обратить ваше внимание на две особенности.

- Создано только четыре строки. Действительно, в таблице **DEPT_AND_EMP** — только четыре строки. 14 строк таблицы **EMP** отдельно не существуют.
- Синтаксис становится весьма экзотичным. Ключевые слова **CAST** и **MULTISET** разработчики обычно никогда не используют. При работе с объектно-реляционными возможностями базы данных придется использовать много экзотических синтаксических конструкций. Ключевое слово **MULTISET** используется, чтобы сообщить серверу Oracle, что подзапрос может вернуть несколько строк (подзапросы в списке выбора оператора **SELECT** ранее могли возвращать только одну строку). Оператор CAST используется, чтобы преобразовать возвращаемое множество в тип набора; в данном случае с помощью **CAST** мы преобразуем многоэлементное множество **(MULTISET)** в данные типа **EMP_TAB_TYPE.** Оператор **CAST** позволяет выполнять преобразование типов в общем случае, не только для наборов. Например, если необходимо извлечь столбец **EMPNO** из таблицы **EMP** как данные типа **VARCHAR2(20),** а **не NUMBER(4),** можно выполнить запрос **select cast(empno as VARCHAR2(20)) e from emp.**

Теперь все готово для запроса данных. Давайте посмотрим, как выглядит одна строка таблицы:

```
tkyte@TKYTE816> select deptno, dname, loc, d.emps AS employees
  2  from dept_and_emp d
  3  where deptno = 10
  4  /
DEPTNO  DNAME          LOC        EMPLOYEES(EMPNO, ENAME, JOB, M

    10  ACCOUNTING     NEW YORK   EMP_TAB_TYPE(EMP_TYPE(7782,
                                   'CLARK', 'MANAGER', 7839,
                                   '09-JUN-81', 2450, NOLL),
                                  EMP_TYPE{7839, 'KING',
                                   'PRESIDENT', NULL,
                                   '17-NOV-81', 5000, NULL),
                                  EMP_TYPE(7934, 'MILLER',
                                   'CLERK', 7782, '23-JAN-82',
                                   1300, NULL))
```

Все данные здесь, в одном столбце. Большинство приложений, если только они не учитывают объектно-реляционные возможности специально, не смогут работать с та-

ким столбцом. Например, интерфейс ODBC не предоставляет средств для работы с вложенными таблицами **(JDBC, OCI, Pro*C, PL/SQL** и большинство других прикладных программных интерфейсов такие средства включают). В подобных случаях сервер Oracle позволяет извлечь вложенный набор и работать с ним, как с обычной реляционной таблицей. Например:

```
tkyte@TKYTE816> select d.deptno, d.dname, emp.*
  2  from dept_and_emp D, table(d.emps) emp
  3  /

DEPTNO DNAME          EMPNO ENAME      JOB              MGR HIREDATE     SAL  COMM
------ -------------- ----- ---------- --------- ---------- --------- ------ -----
    10 ACCOUNTING      7782 CLARK      MANAGER         7839 09-JUN-81   2450
    10 ACCOUNTING      7839 KING       PRESIDENT            17-NOV-81   5000
    10 ACCOUNTING      7934 MILLER     CLERK           7782 23-JAN-82   1300
    20 RESEARCH        7369 SMITH      CLERK           7902 17-DEC-80    800
    20 RESEARCH        7566 JONES      MANAGER         7839 02-APR-81   2975
    20 RESEARCH        7788 SCOTT      ANALYST         7566 09-DEC-82   3000
    20 RESEARCH        7876 ADAMS      CLERK           7788 12-JAN-83   1100
    20 RESEARCH        7902 FORD       ANALYST         7566 03-DEC-81   3000
    30 SALES           7499 ALLEN      SALESMAN        7698 20-FEB-81   1600   300
    30 SALES           7521 WARD       SALESMAN        7698 22-FEB-81   1250   500
    30 SALES           7654 MARTIN     SALESMAN        7698 28-SEP-81   1250  1400
    30 SALES           7698 BLAKE      MANAGER         7839 01-MAY-81   2850
    30 SALES           7844 TURNER     SALESMAN        7698 08-SEP-81   1500     0
    30 SALES           7900 JAMES      CLERK           7698 03-DEC-81    950

14 rows selected.
```

Можно преобразовать столбец **EMPS** в таблицу, и при этом естественно и автоматически произойдет соединение; условия соединения задавать не надо. Поскольку наш тип **EMP** вообще не включает столбец **DEPTNO,** соединять, собственно не по чему. Сервер Oracle учитывает этот нюанс автоматически.

Итак, как же изменить данные? Предположим, необходимо установить комиссионные в объеме 100 долларов всем сотрудникам 10 отдела. Это можно сделать так:

```
tkyte@TKYTE816> update
  2      table(select emps
  3              from dept_and_emp
  4                  where deptno = 10
  5             )
  6  set comm = 100
  7  /

3 rows updated.
```

Вот здесь и вступает в игру "виртуальная таблица для каждой строки". В условии оператора **SELECT,** показанном ранее, было не совсем очевидно, что для каждой строки есть таблица значений, поскольку никаких соединений и прочих подобных конструкций в запросе не было. Все выполнялось как бы чудесным образом. Оператор **UPDATE,** однако, показывает, что для каждой строки есть таблица. Мы выбрали отдельную таб-

лицу для изменения; у этой таблицы нет имени — только идентифицирующий ее запрос. Если бы использовался запрос SELECT, не возвращающий **только одну** таблицу, были бы выданы сообщения об ошибках:

```
tkyte@TKYTE816> update
  2     table(select emps
  3                 from dept_and_emp
  4                   where deptno = 1
  5           )
  6   set com = 100
  7   /
update
*
ERROR at line 1:
ORA-22908: reference to NULL table value


tkyte@TKYTE816> update
  2     table(select emps
  3                 from dept_and_emp
  4                   where deptno > 1
  5           )
  6   set comm = 100
  7 /
  table(select emps
           *
ERROR at line 2:
ORA-01427: single-row subquery returns more than one row
```

Если возвращается менее одной строки (одного экземпляра вложенной таблицы), изменение завершается сообщением об ошибке. Обычно изменение нуля строк — это нормально, но в данном случае выдается такое же сообщение об ошибке, как если бы не было указано имя изменяемой таблицы. Если возвращается несколько строк (более одного экземпляра вложенной таблицы), изменение тоже завершается сообщением об ошибке. Обычно изменение нескольких строк вполне допустимо. Это показывает, что сервер Oracle рассматривает каждую строку в таблице **DEPT_AND_EMP** в качестве указателя на другую таблицу, а не просто набора строк, как в реляционной модели. В этом состоит семантическое отличие вложенной таблицы от пары реляционных таблиц, связанных отношением главная/подчиненная. В случае вложенной таблицы имеется отдельная таблица для каждой родительской строки. Это различие иногда может усложнить работу с вложенными таблицами. Рассмотрим используемую модель, наглядно представляющую данные для одного отдела. Но она абсолютно не подходит для запросов типа: "В каком отделе работает **KING?",** "Сколько бухгалтеров работает в организации?" и тому подобных. Эти запросы лучше выполнять к реляционной таблице EMP, но в нашей модели с вложенной таблицей добраться до данных таблицы **EMP** можно только через данные таблицы DEPT. Всегда надо выполнять соединение — нельзя запросить данные только из таблицы **EMP.** Сделать это стандартным и описанным в документации способом нельзя, но можно использовать один трюк (подробнее о нем — поз-

же). Если необходимо изменить все строки в таблице EMPS_NT, придется выполнить 4 изменения (отдельно для каждой строки в таблице **DEPT_AND_EMP),** чтобы изменить виртуальную таблицу, связанную с каждой строкой.

Необходимо также учесть, что при изменении данных о сотруднике отдела 10, семантически происходит изменение столбца **EMPS** таблицы **DEPT_AND_EMP.** Физически используется две таблицы, но семантически есть только одна. Хотя в таблице отделов данные не изменялись, строка, содержащая измененную вложенную таблицу, блокируется и не может быть изменена другими сеансами. При традиционной взаимосвязи главная/подчиненная такой проблемы нет.

Вот почему я стараюсь не использовать вложенные таблицы для постоянного хранения данных. Лишь к **немногим** подчиненным таблицам запросы отдельно не выполняются. В рассмотренном примере таблица **EMP** — отдельная, самостоятельная сущность и должна запрашиваться отдельно. Так бывает практически всегда. Я обычно использую вложенные таблицы в представлениях на основе реляционных таблиц. Подробно мы будем рассматривать это в главе 20, посвященной объектно-реляционным возможностям.

Теперь, когда вы знаете, как изменять экземпляр вложенной таблицы, вставка и удаление из нее строк не составит трудностей. Давайте добавим строку в экземпляр вложенной таблицы для отдела 10 и удалим одну из строк для отдела 20:

```
tkyte@TKYTE816> insert into table
  2   (select emps from dept_and_emp where deptno = 10)
  3   values
  4   (1234, 'NewEmp', 'CLERK', 7782, sysdate, 1200, null);

1 row created.

tkyte@TKYTE816> delete from table
  2  (select emps from dept_and_emp where deptno = 20)
  3 where ename = 'SCOTT';

1 row deleted.

tkyte@TKYTE816> select d.dname, e.empno, ename
  2   from dept_and_emp d, table(d.emps) e
  3   where d.deptno in (10, 20);

DNAME          EMPNO  ENAME

RESEARCH        7369  SMITH
RESEARCH        7566  JONES
RESEARCH        7876  ADAMS
RESEARCH        7902  FORD
ACCOUNTING      7782  CLARK
ACCOUNTING      7839  KING
ACCOUNTING      7934  MILLER
ACCOUNTING      1234  NewEmp

8 rows selected.
```

Итак, вот базовый синтаксис операторов для запросов и изменения вложенных таблиц. Оказывается, для использования вложенные таблицы часто приходится извлекать (как пришлось это делать мне в рассмотренных примерах), особенно при выполнении запросов. Усвоив концепцию "виртуальная таблица для каждой строки", вы упростите себе работу с вложенными таблицами.

Сначала я категорически утверждал: "Необходимо всегда выполнять соединение, нельзя получить данные только из таблицы **EMP",** а затем смягчился: "Вообще-то можно, если очень хочется". Этот метод не описан в документации и не поддерживается, так что используйте его только как **последнюю надежду,** если ничего другого не остается. Наиболее удобен он в случаях, когда необходимо выполнить множественное изменение данных во вложенной таблице (помните, нам приходилось делать это через таблицу **DEPT** путем соединения). Есть недокументированная подсказка, **NESTED_TABLE_GET_REFS,** используемая утилитами **EXP** и **IMP** для работы с вложенными таблицами. Ее использование также позволяет лучше понять физическую структуру вложенных таблиц. Эту "волшебную" подсказку легко обнаружить после экспортирования таблицы с вложенной таблицей. Я экспортировал представленную выше таблицу, чтобы получить ее расширенное определение с помощью утилиты **IMP.** После экспортирования в разделяемом пуле (в представлении **V$SQL)** я обнаружил следующий SQL-оператор:

```
SELECT /*+NESTED_TABLE_GET_REFS+*/ NESTED_TABLE_ID, SYS_HC_ROWINFO$ FROM
   "TKYTE"."EMPS_NT"
```

Найти его позволил простой запрос вида **SELECT SQL_TEXT FROM V$SQL WHERE UPPER(SQL_TEXT) LIKE '%EMP%'.** Если выполнить найденный SQL-оператор, можно получить "волшебные" результаты:

```
tkyte@TKYTE816> SELECT /*+NESTED_TABLE_GET_REFS+*/
  2           NESTED_TABLE_ID, SYS_NC_ROWINFO$
  3  FROM "TKYTE"."EMPS_NT"
  4  /

NESTED_TABLE_ID                    SYS_NC_ROWINFO$(EMPNO, ENAME,

9A39835005B149859735617476C9A80E   EMP_TYPE(7782,  'CLARK',
                                   'MANAGER', 7839, '09-JUN-81',
                                   2450, 100)

9A39835005B149859735617476C9A80E   EMP_TYPE(7839,  'KING',
                                   'PRESIDENT', NULL,
                                   '17-NOV-81',  5000,  100)
```

Да, несколько удивительно, ведь если получить описание таблицы:

```
tkyte@TKYTE816> desc enps_nt
 Name                      Null?  Type

 EMPNO                            NUMBER(4)
 ENAME                            VARCHAR2(10)
 JOB                              VARCHAR2(9)
 MGR                              NUMBER(4)
```

```
HIREDATE                       DATE
SAL                            NUMBER(7,2)
COMM                           NUMBER(7,2)
```

этих двух столбцов в нем вообще нет. Они являются частью скрытой реализации вложенных таблиц. Столбец NESTED_TABLE_ID фактически является внешним ключом к главной таблице DEPT_AND_EMP. Таблица DEPT_AND_EMP имеет скрытый столбец, используемый для соединения с таблицей EMPS_NT. "Столбец" SYS_NC_ROWINF$ — магический. Это скорее функция, а не столбец. Вложенная таблица в данном случае является объектной (она построена по объектному типу), а SYS_NC_INFO$ — внутренний способ ссылки на строку как на объект (вместо отдельных ссылок на все реляционные столбцы) в базе данных Oracle. "За кулисами" сервер Oracle автоматически реализовал отношение главная/подчиненная между таблицами, с первичный и внешним ключами, сгенерированными системой. Если покопаться еше, можно добраться до "реального" словаря данных, и получить информацию обо всех столбцах таблицы DEPT_AND_EMP:

```
tkyte@TKYTE816> select name
  2     from sys.col$
  3    where obj# = (select object_id
  4                    from user_objects
  5                   where object_name = 'DEPT_AKD_EMP')
  6   /

NAME
------------------------------
DEPTNO
DNAME
LOC
EMPS
SYS_NC0000400005$


tkyte@TKYTE816> select SYS_NC0000400005$ from dept_and_emp;

SYS_NC0000400005$

9A39835005B149859735617476C9A80E
A7140089B1954B39B73347EC20190D68
20D4AA0839FB49B0975FBDE367842E16
56350C866BA24ADE8CF9E47073C52296
```

Столбец со странным именем SYS_NC0000400005$ — это сгенерированный системой ключ для таблицы DEPT_AND_EMP. Если продолжить исследование, можно выяснить, что сервер Oracle создал по этому столбцу уникальный индекс. К сожалению, однако, он не проиндексировал столбец NESTED_TABLE_ID в таблице EMPS_NT. Этот столбец надо проиндексировать, поскольку всегда выполняется соединение из таблицы DEPT_AND_EMP с таблицей EMPS_NT. Об этом важно помнить при использовании стандартных вложенных таблиц, как в рассмотренном ранее примере: всегда индексируйте столбец NESTED_TABLE_ID вложенных таблиц!

Я немного отвлекся от темы. Итак, мы говорили о том, каким образом можно работать с вложенной таблицей, как с обычной. Это делается с помощью подсказки NESTED_TABLE_GET_REFS. Ее можно использовать так:

```
tkyte@TKYTE816> select /*+ nested table_get_refs */ empno, ename
  2  front emps_nt where ename like '%A%' ;

  EMPNO  ENAME
-------  ----------
   7782  CLARK
   7876  ADAMS
   7499  ALLEN
   7521  WARD
   7654  MARTIN
   7698  BLAKE
   7900  JAMES
7 rows selected.

tkyte@TKYTE816> update /*+ nested_table_get_refs */ emps_nt
  2  set ename = initcap(ename);

14 rows updated.

tkyte@TKYTE816> select /*+ nested_table_get_refs */ empno, ename
  2  from emps_nt where ename like '%a%';

  EMPKO  ENAME
-------  ----------
   7782  Clark
   7876  Adams
   7521  Ward
   7654  Martin
   7698  Blake
   7900  James
6 rows selected.
```

Повторю еще раз: эта возможность не отражена в документации и официально не поддерживается. Она может использоваться не во всех средах. Речь идет о специфической функциональной возможности, обеспечивающей работу утилит EXP и IMP. Это единственная среда, где она гарантированно работает. Используйте эту подсказку на свой страх и риск. Используйте, однако, осторожно, и не включайте в производственный код. Используйте ее для разовых исправлений данных или для получения содержимого вложенной таблицы. Официально поддерживается только извлечение данных столбца в виде таблицы следующим образом:

```
tkyte@TKYTE816> select d.deptno, d.dname, emp.*
  2  from dept_and_emp D, table(d.emps) emp
  3  /
```

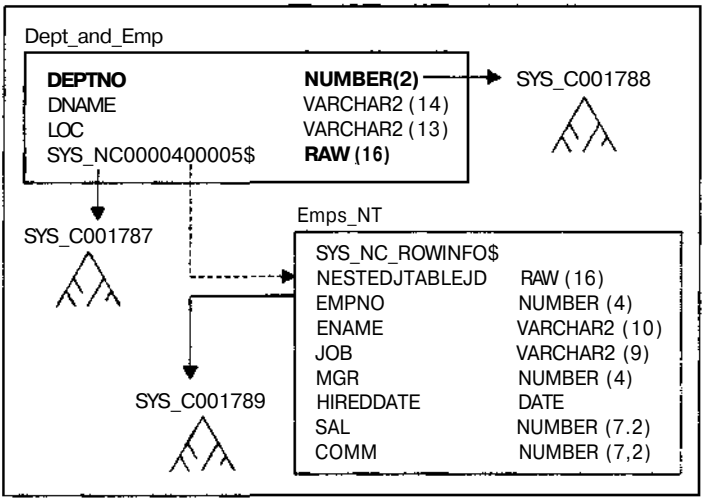
Именно этот прием следует использовать в запросах и в производственном коде.

Хранение вложенных таблиц

Кое-что о том, как хранится вложенная таблица, нам уже известно. Сейчас мы чуть глубже рассмотрим стандартно создаваемую сервером Oracle структуру, а также параметры, которыми можно управлять. Вернемся к рассмотренному ранее оператору создания таблицы:

```
tkyte@TKYTE816> create table dept_and_emp
  2   (deptno number(2) primary key,
  3    dname   varchar2(14),
  4    loc           varchar2(13),
  5    emps           emp_tab_type
  6  )
  7  nested table emps store as emps_nt;

Table created.

tkyte@TKYTE816> alter table emps_nt add constraint emps_empno_unique
  2                 unique(empno)
  3  /
Table altered.
```

Мы знаем, что фактически сервер Oracle создает следующую структуру:



Создается две таблицы. Таблица, создаваемая явно, получила дополнительный скрытый столбец (по умолчанию скрытый столбец будет создаваться для **каждого** столбца типа вложенной таблицы). По этому скрытому столбцу создается также требование **уникальности.** Сервер Oracle автоматически создал вложенную таблицу **EMPS_NT.** Эта таблица включает два скрытых столбца, причем один из них, **SYS_NC_ROWINFO$,** — виртуальный; он возвращает все скалярные элементы в виде объекта. Второй столбец - внешний ключ, **NESTED_TABLE_ID,** по которому вложенную таблицу можно соеди-

нять с главной. Обратите внимание на **отсутствие** индекса по этому столбцу! Наконец, сервер Oracle добавил индекс по столбцу **DEPTNO** в таблице **DEPT_AND_EMP** для ограничения первичного ключа. Итак, мы хотели создать таблицу, но получили намного больше. Если разобраться, создаются практически те же структуры, что и для поддержки отношения главная/подчиненная таблица, но в последнем случае мы бы использовали существующий первичный ключ по столбцу **DEPTNO** в качестве внешнего ключа в таблице EMPS_NT, а не генерировали бы суррогатный ключ типа **RAW(16).**

Если поинтересоваться, как сохраняют нашу вложенную таблицу утилиты **EXP/IMP,** можно увидеть следующее:

```
CREATE TABLE "TKYTE"."DEPT_AND_EMP"
 ("DEPTNO" NUMBER(2, 0),
  "DKAME"  VARCHAR2(14),
  "LOC"     VARCHAR2(13),
  "EMPS" "EMP_TAB_TYPE")
PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 LOGGING
STORAGE(INITIAL 131072 NEXT 131072
          MINEXTENTS 1 MAXEXTENTS 4096
          PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
          BUFFER_POOL DEFAULT)
TABLESPACE "USERS"
NESTED TABLE "EMPS"
    STORE AS "EMPS_NT"
    RETURN AS VALUE
```

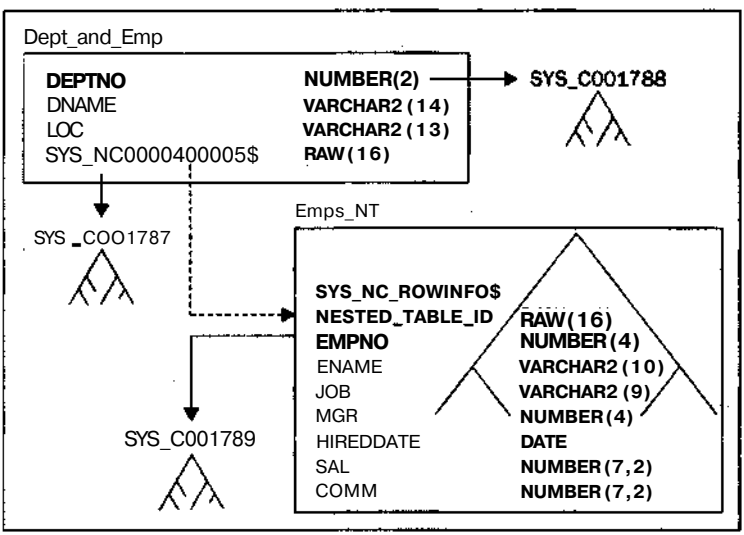
Единственная новая конструкция здесь — RETURN AS VALUE. Она описывает, как вложенная таблица возвращается клиентскому приложению. По умолчанию сервер Oracle передает вложенную таблицу клиенту по значению: вместе с каждой строкой передаются фактические данные. Можно также задать **RETURN AS LOCATOR,** что означает возвращать клиенту указатель на данные (локатор), а не сами данные. Данные будут передаваться только после того, как клиент разыменует этот указатель. Поэтому, если предполагается, что клиент не будет просматривать строки вложенной таблицы для каждой строки основной таблицы, можно возвращать указатель, а не значения, уменьшая объем передаваемой по сети информации. Например, если клиентское приложение отображает список отделов, а при двойном щелчке на строке отдела — информацию о его сотрудниках, имеет смысл использовать указатель. Детальная информация о сотрудниках обычно не просматривается. Но этот случай скорее исключение, чем правило.

Что еще можно сделать с вложенной таблицей? Во-первых, столбец **NESTED_TABLE_ID** необходимо проиндексировать. Поскольку к вложенной таблице обращаются из главной как к подчиненной, этот индекс действительно необходим. Можно проиндексировать этот столбец с помощью оператора CREATE INDEX, но лучше хранить вложенную таблицу как организованную по индексу. Вложенная таблица — еще один прекрасный пример использования таблицы, организованной по индексу. При этом все подчиненные строки будут физически объединяться по значению столбца NESTED_TABLE_ID (так что извлечение данных потребует меньше операций физического ввода/вывода). При этом также не нужно создавать дополнительный индекс по столбцу **RAW(16).** Если оптимизировать дальше, то, поскольку столбец

NESTED_TABLE_ID будет первым в первичном ключе таблицы, организованной по индексу, можно также включить сжатие ключей индекса, чтобы не хранить избыточные значения NESTED_TABLE_ID. Кроме того, можно включить требования UNIQUE и NOT NULL для столбца EMPNO сразу в текст оператора CREATE TABLE. Поэтому я немного изменю представленный выше оператор CREATE TABLE:

```
CREATE TABLE "TKYTE"."DEPT_AND_EMP"
 ("DEPTNO" NUMBER(2, 0),
  "DNAME"  VARCHAR2(14),
  "LOC"    VARCHAR2(13) ,
"EMPS" "EMP_TAB_TYPE")
PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 LOGGING
STORAGE(INITIAL 131072 NEXT 131072
          MINEXTENTS 1 MAXEXTENTS 4096
          PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
          BUFFER_POOL DEFAULT)
TABLESPACE "USERS"
NESTED TABLE "EMPS"
    STORE AS "EMPS_NT"
    ((empno NOT NULL, unique (empno) , primary key (neated_table_id,einpno))
      organization index congress 1)
    RETURN AS VALUE
/
```

Теперь мы получим следующий набор объектов. Вместо обычной таблицы EMP_NT теперь имеется организованная по индексу таблица EMPS_NT, что и показано на следующей схеме:



Поскольку **EMPS_NT** — организованная по индексу таблица со сжатием, она занимает меньше места, чем исходная стандартная вложенная таблица, причем **сразу** имеется необходимый индекс.

В завершение разговора о вложенных таблицах должен сказать, что лично я не использую их для постоянного хранения данных по следующим причинам.

- Дополнительные расходы ресурсов на добавляемые столбцы типа RAW(16). Эти дополнительные столбцы создаются как в главной, так и в подчиненной таблице. В главной таблице будет по дополнительному столбцу RAW(16) для каждого ее столбца типа вложенной таблицы. Поскольку в главной таблице обычно есть первичный ключ (в моих примерах — DEPTNO), имеет смысл использовать в подчиненных таблицах именно его, а не сгенерированный системой ключ.
- Дополнительные расходы ресурсов на поддержку требования уникальности для главной таблицы, и так обычно имеющей требование уникальности.
- Саму вложенную таблицу не так просто использовать, если не применять конструкции, официально не поддерживаемые корпорацией Oracle (вроде подсказки NESTED_TABLE_GET_REFS). Ее можно использовать явно в запросах, но не при множественных изменениях.

Однако я активно использую вложенные таблицы при программировании и в представлениях. Именно там, я думаю, они наиболее уместны; в главе 20 я продемонстрирую, как их использовать в этих случаях. В качестве механизма хранения данных я предпочитаю явное создание главной и подчиненной таблиц. Создав главную и подчиненную таблицы, можно создать представление и работать с ним так, как с вложенной таблицей. Другими словами, можно получить все преимущества вложенной таблицы без дополнительных расходов ресурсов. Как это сделать, будет описано в главе 20, посвященной использованию объектно-реляционных средств.

Если вложенные таблицы все же используются как способ хранения, не забудьте организовать вложенную таблицу по индексу, чтобы избежать дополнительных расходов ресурсов на поддержку отдельного индекса по столбцу NESTED_TABLE_ID, помимо таблицы. Советы по созданию таблиц, организованных по индексу, конфигурированию сегмента остатка и использованию других возможных опций можно найти в предыдущем разделе. Если не используете организацию таблицы по индексу, не забудьте создать индекс по столбцу NESTED_TABLE_ID вложенной таблицы, чтобы избежать ее полного просмотра при поиске подчиненных строк.

Временные таблицы

Временные таблицы используются для хранения промежуточных результирующих множеств на время транзакции или сеанса. Хранящиеся во временной таблице данные доступны только текущему сеансу; сеансу недоступны данные другого сеанса, даже если они зафиксированы. Одновременный доступ нескольких пользователей при работе с временными таблицами тоже не проблема: при использовании временной таблицы сеансы никогда не блокируют друг друга. Даже если "заблокировать" временную таблицу, это не помешает другим сеансам использовать "свои" временные таблицы. Как было показано в главе 5, при работе с временными таблицами генерируется меньше информации повторного выполнения, чем при работе с обычными таблицами. Однако, поскольку для

содержащихся в этих таблицах данных необходимо генерировать данные отмены, определенная информация в журнал повторного выполнения все же поступает. Больше всего их будет сгенерировано при выполнении операторов UPDATE и DELETE; операторы INSERT и SELECT генерируют таких данных намного меньше.

Память под временные таблицы выделяется из временного табличного пространства подключившегося пользователя или, если к ним обращаются из процедур, работающих с правами создателя, — из временного табличного пространства владельца процедуры. Глобальная временная таблица — всего лишь шаблон для создания таблицы. Создание временной таблицы не требует выделения пространства; первоначальный (INITIAL) экстент, как для обычной таблицы, не выделяется. Вместо этого по ходу работы (при. первом добавлении данных во временную таблицу) создается временный сегмент для сеанса. Поскольку каждый сеанс получает собственный временный сегмент (а не просто экстент в существующем сегменте), пользователь может выделять пространство под временную таблицу в другом табличном пространстве. Пользователь USER1 может использовать временное табличное пространство TEMPI — временные таблицы будут размещаться в этом пространстве. Временные таблицы пользователя USER2 будут размещаться во временном табличном пространстве TEMP2.

Временные таблицы в Oracle подобны временным таблицам в других реляционных СУБД, но определяются статически, т.е. создаются в базе данных один раз, а не в каждом сеансе или хранимой процедуре. Они существуют всегда, и будут храниться в словаре данных как объекты, но будут казаться пустыми, пока сеанс не поместит в них данные. Тот факт, что они определены статически, позволяет создавать представления, ссылающиеся на временные таблицы, создавать хранимые процедуры, использующие для работы с ними статические операторы SQL, и т.д.

Временные таблицы могут создаваться *на время сеанса* (данные остаются в таблице при фиксации транзакций, но исчезают при завершении сеанса). Их также можно создавать *на время транзакции* (данные исчезают после завершения транзакции). Вот пример, демонстрирующий особенности временных таблиц обоих видов. В качестве шаблона и использовал таблицу SCOTT.EMP:

```
tkyte@TKYTE816> create global temporary table temp_table_session
  2  on commit preserve rows
  3  as
  4  select * from scott.emp where 1=0
  5  /

Table created.
```

Конструкция ON COMMIT PRESERVE ROWS означает, что временная таблица создается на время сеанса. Строки останутся в таблице, пока не завершится сеанс или пока они не будут удалены явно с помощью операторов DELETE или TRUNCATE. Эти строки видны только в сеансе, их создавшем; в другом сеансе они не будут видны даже после выполнения оператора COMMIT:

```
tkyte@TKYTE816> create global temporary table temp_table_transaction
  2  on commit delete rows
  3  as
```

```
  4   select * from scott.emp where 1=0
  5   /

Table created.
```

Конструкция ON COMMIT DELETE ROWS означает, что временная таблица создается на время транзакции. Когда транзакция завершается, ее строки исчезают, поскольку выделенные таблице временные экстенты освобождаются. Никаких дополнительных расходов ресурсов на автоматическую очистку временных таблиц не требуется. Теперь давайте рассмотрим отличия между этими двумя типами.

```
tkyte@TKYTE816> insert into temp_table_seseion select * from scott.emp;

14 rows created.

tkyte@TKYTE816> insert into temp_table_transaction select * from scott.emp;

14 rows created.
```

Мы только что поместили 14 строк в каждую из временных таблиц, и, как показывает следующий оператор, мы их "видим":

```
tkyte@TKYTE816> select session_cnt, transaction_cnt
  2     from (select count(*) session_cnt from temp_table_session),
  3          (select count(*) transaction_cnt from temp_table_transaction);

  SESSION CNT    TRANSACTION CNT
-------------  -----------------
           14                 14

tkyte@TKYTE816> commit;
```

Поскольку транзакция зафиксирована, мы увидим строки в таблице, созданной на время сеанса, но таблица, созданная на время транзакции, будет пустой:

```
tkyte@TKYTE816> select session_cnt, transaction_cnt
  2     from (select count(*) session_cnt from terop_table_session),
  3          (select count(*) txansaction_cnt from temp_table_transaction);

   SESSION CNT    TRANSACTION CNT
-------------  -----------------
            14                  0

tkyte@TKYTE816> disconnect
Disconnected from 0racle8i Enterprise Edition Release 8.1.6.0.0 — Production
With the Partitioning option
JServer Release 8.1.6.0.0 — Production
tkyte@TKYTE816> connect tkyte/tkyte
Connected.
```

Поскольку мы создали новый сеанс, ни в одной из таблиц данных не будет:

```
tkyte@TKYTE816> select session_cnt, transaction_cnt
  2     from (select count(*) session_cnt from temp_table_session),
  3          (select count(*) transaction_cnt from temp_table_transaction);
```

```
   SESSION CNT     TRANSACTION CNT
-------------- -----------------
             0                   0
```

Если у вас есть опыт работы с временными таблицами в СУБД SQL Server и/или Sybase, главное — учесть, что вместо выполнения оператора **select x, y, *z* into #temp from некая_таблица** для динамического создания и наполнения данными временной таблицы, необходимо следующее.

- Один раз в базе данных создать все таблицу TEMP как глобальную временную. Это делается при установке приложения, аналогично созданию обычных таблиц.
- В процедуре выполнить оператор **insert into temp (x, y, z) select x, y, z from некая_таблица.**

Подчеркну еще раз — цель не в том, чтобы создавать таблицы в хранимых процедурах по ходу выполнения. Этот способ для СУБД Oracle не подходит. Выполнение оператора ЯОД — действие дорогостоящее, и при работе процедур его надо избегать. Временные таблицы для приложения должны создаваться при его установке, а **не** по ходу работы.

Временные таблицы могут иметь многие атрибуты обычных таблиц. Для них можно задавать триггеры, ограничения целостности, создавать индексы и т.д. Не поддерживаются следующие возможности обычных таблиц:

- нельзя задавать требования целостности ссылок — временные таблицы не могут быть **целевыми** для внешнего ключа и для них нельзя задавать требование внешнего ключа;
- нельзя использовать столбцы типа **VARRAY** и NESTED TABLE;
- нельзя организовывать временные таблицы по индексу;
- нельзя размещать временные таблицы в индексном или хеш-кластере;
- нельзя фрагментировать временные таблицы;
- для них нельзя сгенерировать статистическую информацию с помощью оператора ANALYZE.

Один из недостатков временных таблиц в базе данных состоит в том, что оптимизатор не имеет по ним реальной статистической информации. При использовании *оптимизатора основанного на стоимости* (Cost-Based Optimizer — CBO), актуальная статистическая информация принципиально важна для успешной оптимизации. При отсутствии статистической информации оптимизатор будет делать предположения о распределении и объеме данных, а также избирательности индекса. Если эти предположения окажутся неверными, планы выполнения, сгенерированные для запросов, интенсивно использующих временные таблицы, окажутся далекими от оптимальности. Во многих случаях, правильное решение состоит в том, чтобы вообще не использовать временную таблицу, а использовать вместо нее *подставляемое представление* (inline view). Пример использования подставляемого представления можно найти в последнем из показанных выше операторов **SELECT** (их там используется два). При этом серверу Oracle доступна вся

необходимая статистическая информация о таблице, и он сможет построить оптимальный план выполнения запроса.

Часто приходится видеть, как разработчики используют временные таблицы, усвоив при работе с другими СУБД, что соединение слишком большого количества таблиц в одном запросе — это "плохо". От этой привычки при работе с СУБД Oracle надо избавляться. Лучше не пытаться превзойти оптимизатор, разбивая один запрос на три или четыре запроса, хранящих промежуточные результаты во временных таблицах, и затем соединяя эти временные таблицы, а просто использовать один запрос, извлекающий необходимую информацию. Ссылаться на множество таблиц в одном запросе — вполне допустимо; обходные пути с использованием временных таблиц для таких запросов в СУБД Oracle не нужны.

В других случаях, однако, использование временной таблицы бывает вполне оправдано. Например, я недавно написал приложение для наладонника Palm, синхронизирующее дневник Palm Pilot с календарной информацией, хранящейся в базе данных Oracle. Palm выдает мне список всех записей, изменившихся с момента последней синхронизации. Я должен был получить эти записи и сравнить с текущими данными в базе, изменить записи базы данных и затем сгенерировать список изменений, которые надо применить к информации в Palm. Это прекрасный пример того, когда временная таблица особенно полезна. Я использовал временную таблицу для хранения в базе данных изменений информации, выполненных в Palm. Затем выполнил хранимую процедуру, сравнивающую сгенерированные наладонником изменения с постоянно изменяющимися (и очень большими) постоянными таблицами, чтобы определить, какие изменения необходимо выполнить в данных СУБД Oracle, а какие — скопировать с СУБД Oracle в Palm. По этим данным приходится выполнять несколько проходов. Сначала я нахожу все записи, которые были изменены только на Palm и делаю соответствующие изменения в базе данных Oracle. Затем нахожу все записи, измененные с момента последней синхронизации как на Palm, так и в базе данных, и разбираюсь с ними. После этого я нахожу все записи, измененные только в базе данных, и вношу соответствующие изменения во временную таблицу. Наконец, программа синхронизации Palm выбирает изменения из временной таблицы и применяет их к информации наладонника. После отключения временные данные исчезают.

Я, однако, столкнулся с проблемой: поскольку постоянные таблицы были проанализированы, использовался оптимизатор, основанный на стоимости. Статистической информации по временной таблице не было (проанализировать временную таблицу можно, но статистическая информация при этом не собирается), и оптимизатор, основанный на стоимости делал о ней множество "предположений". Я как разработчик знал среднее количество строк в этой таблице, распределение данных, избирательность индексов и т.п. Мне нужен был способ сообщить оптимизатору эти, **более точные,** сведения. Это легко сделать с помощью пакета **DBMS_STATS.**

Поскольку оператор **ANALYZE** не собирает статистическую информацию о временной таблице, необходимо вручную заполнить словарь данных репрезентативной информацией о временных таблицах. Например, если в среднем в таблице будет 500 строк, со средним размером 100 байт и таблица будет занимать примерно 7 блоков, можно использовать следующий вызов:

```
tkyte@TKYTE816> begin
  2      dbms_stats.set_table_stats (ownname => USER,
  3                          tabname  => 'T',
  4                          numrows  => 500,
  5                          numblks  => 7,
  6                          avgrlen  => 100) ;
  7 end;
  8  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select table_name, num_rows, blocks, avg_row_len
  2                   from user_tables
  3                  where table_name = 'T';

TABLE_NAME              NUM_ROWS     BLOCKS     AVG_ROW_LEN
--------------------- ---------- ---------- ---------------
T                          500          7             100
```

Теперь оптимизатор больше не будет использовать свои предположения — он будет использовать **наши.** Если продолжить в том же духе, можно заставить сервер использовать куда более точную статистическую информацию. Следующий пример показывает использование временной таблицы оптимизатором, основанным на стоимости. План выполнения запроса, сгенерированный без учета статистической информации, — не оптимален. Оптимизатор "решил" использовать индекс, хотя этого не надо было делать. Такой план был выбран, исходя из стандартной информации об избирательности индекса, количества строк в таблице и выбираемого количества строк и т.п. Чтобы исправить это, я ненадолго удалил временную таблицу, создал постоянную таблицу с тем же именем и структурой и поместил в нее ренрезентативные данные. Затем я тщательно проанализировал таблицу (я мог бы также сгенерировать гистограммы и т.п.) и с помощью пакета DBMS_STATS экспортировал статистическую информацию для этой постоянной таблицы. Затем я удалил постоянную таблицу и пересоздал временную. После этого осталось только импортировать репрезентативную статистическую информацию, и оптимизатор выбрал правильный план:

```
tkyte@TKYTE816> create global temporary table temp_all_objects
  2  as
  3  select * from all_objects where 1=0
  4  /

Table created.

tkyte@TKYTE816> create index temp_all_objects_idx on
temp_all_objects(object_id)
  2  /

Index created.

tkyte@TKYTE816> insert into temp_all_objects
  2   select * from all_objects where rownum < 51
  3  /

50 rows created.
```

```
tkyte@TKYTE816> set autotrace on explain
tkyte@TKYTE816> select /*+ ALL_ROWS */ object_type, count(*)
  2     FROM temp_all_objects
  3    where object_id < 50000
  4     group by object_type
  5   /

OBJECT TYPE        COUNT(*)
---------------- -----------
JAVA CLASS              50


Execution  Plan
----------------------------------------------------------------------------
    0      SELECT STATEMENT Optimizer=HINT: ALL_ROWS  (Cost=13 Card=409
    1    0    SORT  (GROUP BY)  (Cost=13 Card=409 Bytes=9816)
    2    1      TABLE ACCESS  (BY INDEX ROWID)  OF  'TEMP_ALL_OBJECTS'  (Cost=10
    3    2        INDEX  (RANGE SCAN)  OF  'TEMP_ALL_OBJECTS_IDX'  (NON-UNIQUE)

tkyte@TKYTE816> set autotrace off
```

Это показывает, что оптимизатор, основанный на стоимости, выбрал неверный план. При обращении к более чем 10-20 процентам таблицы индекс использовать не стоит. Мы же обращаемся ко всей таблице; и таблица настолько маленькая, что использование индекса ничего не даст в любом случае. Вот как предоставить оптимизатору информацию, необходимую для выработки правильного плана:

```
tkyte@TKYTE816> drop table temp_all_objects;

Table dropped.

tkyte@TKYTE816> create table temp_all_objects
  2   as
  3   select * from all_objects where 1=0
  4   /

Table created.

tkyte@TKYTE816> create index temp_all_objects_idx on
temp_all_objects(object_id)
  2   /

Index created.

tkyte@TKYTE816> insert into temp_all_objects
  2   select * from all_objects where rownum < 51;

50 rows created.

tkyte@TKYTE816> analyze table temp_all_objects compute statistics;

Table analyzed.
tkyte@TKYTE816> analyze table temp_all_objects compute statistics for all
  2   indexes;

Table analyzed.
```

Я всего лишь создал постоянную таблицу, аналогичную временной и наполнил ее репрезентативными данными. В этом — весь фокус; необходимо хорошо продумать, какие данные помешаются в таблицу перед анализом. Предположения оптимизатора будут заменены этими данными, так что надо предоставить ему данные поточнее, чем он строит сам. В некоторых случаях бывает достаточно просто установить статистическую информацию о таблице или индексе вручную, как было сделано в предыдущем примере, чтобы сообщить оптимизатору количество и диапазон значений. В других случаях, чтобы дать оптимизатору, основанному на стоимости, необходимые данные, придется добавить в словарь данных много разнообразной информации. Можно не добавлять ее вручную, а поручить это серверу Oracle. Представленный ниже метод позволяет получить и легко задать всю информацию:

```
tkyte@TKYTE816> begin
  2    dbms_stats.create_stat_table    (ownname => USER,
  3                                     stattab =>  'STATS');
  4
  5       dbms_stats.export_table_stats (ownname => USER,
  6                   tabname                    =>  'TEMP_ALL_OBJECTS' ,
  7                                     stattab =>  'STATS') ;
  8       dbms_stats.export_index_stats (ownname => USER,
  9                                     indname =>  'TEMP_ALL_OBJECTS_IDX' ,
 10                                     stattab =>  'STATS');
 11  end;
 12  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> drop table temp_all_objects;
Table dropped.

tkyte@TKYTE816> create global temporary table temp_all_objects
  2  as
  3  select * from all_objects where 1=0
  4  /

Table created.

tkyte@TKYTE816> create index temp_all_objects_idx on
temp_all_objects(object_id)
  2  /

Index created.

tkyte@TKYTE816> begin
  2       dbms_stats.import_table_stats(ownname => USER,
  3                               tabname =>  'TEMP_ALL_OBJECTS',
  4                               stattab =>  'STATS');
  5       dbms_stats.import_index_stats(ownname => USER,
  6                               indname =>  'TEMP_ALL_OBJECTS_IDX',
  7                               stattab =>  'STATS');
  8  end;
  9  /

PL/SQL procedure successfully completed.
```

Мы просто добавили статистическую информацию для временной таблицы на основе репрезентативного набора данных. Теперь оптимизатор, основанный на стоимости, будет использовать ее для выбора оптимального плана, что подтверждает следующий запрос:

```
tkyte@TKYTE816> insert into temp_all_objects
  2   select * from all_objects where rownum < 51
  3   /

50 rows created.

tkyte@TKYTE816> set autotrace on
tkyte@TKYTE816> select /*+ ALL_ROWS */ object_type, count(*)
  2      FROM temp_all_objects
  3    where object_id < 50000
  4      group by object_type
  5   /

OBJECT_TYPE             COUNT(*)
----------------- -----------
JAVA CLASS                   50

Execution Plan
----------------------------------------------------------------------------
    0       SELECT STATEMENT Optimizer=HINT: ALLJROWS (Cost*3 Card=l Bytes=14)
    1    0    SORT (GROUP BY) (Cost=3 Card=l Bytes=14)
    2    1      TABLE ACCESS (FULL) OF 'TEMP_ALL_OBJECTS' (Cost=l Card=50
```

Временные таблицы могут пригодиться в приложении, где необходимо временно, на период сеанса или транзакции, запомнить набор строк для соединения с другими таблицами. Не предполагается их использование для разбиения большого запроса на меньшие результирующие множества для последующего соединения (для этого чаще всего и используются временные таблицы в других СУБД). В СУБД Oracle запрос, разбитый на меньшие, записывающие промежуточные результаты во временные таблицы, как правило, выполняется медленнее. Я постоянно убеждаюсь в этом, когда, переписав ряд вставок во временные таблицы результатов нескольких запросов в виде одного большого запроса, получаю результат намного быстрее.

При работе с временными таблицами генерируется минимальный объем данных повторного выполнения, но данные эти все же генерируются, и избежать этого нельзя. Данные повторного выполнения генерируются для данных отмены и обычно их объем пренебрежимо мал. Если к временным таблицам применяются только операторы **INSERT** и SELECT, генерация данных повторного выполнения будет незаметна (этих данных генерируется мало). Большой объем данных повторного выполнения будет генерироваться только при интенсивном удалении или изменении строк во временной таблице.

Статистическая информация для оптимизатора, основанного на стоимости, для временной таблицы не генерируется, но ее можно задать с помощью пакета **DBMS_STATS.** Можно установить лишь несколько наиболее существенных показателей (количество строк, среднюю длину строки и т.п.) или использовать для генерации полного набора статистической информации постоянную таблицу с репрезентативными данными. Убе-

дитесь только, что ваши предположения лучше стандартных. В противном случае генерируемый оптимизатором план только ухудшится.

Объектные таблицы

Мы уже касались того, как работать с объектной таблицей, в разделе, посвященном вложенным таблицам. *Объектной* называется таблица, создаваемая на основе типа, а не набора столбцов. Обычно оператор CREATE TABLE выглядит следующим образом:

```
create table t (x int, y date, z varchar2(25));
```

Оператор создания объектной таблицы имеет следующий вид:

```
create table t of Some_Type;
```

Атрибуты (столбцы) таблицы t берутся из определения типа SOME_TYPE. Давайте рассмотрим простой пример, в котором используется несколько типов, и разберем полученные структуры данных:

```
tkyte@TKYTE816> create or replace type address_type
  2  as object
  3  (city     varchar2(30),
  4   street   varchar2(30),
  5   state    varchar2(2),
  6   zip      number
  7  )
  8  /

Type created.

tkyte@TKYTE816> create or replace type person_type
  2  as object
  3  (name           varchar2(30) ,
  4   dob             date,
  5   home_address    address_type,
  6   work_address    address_type
  7  )
  8  /

Type created.

tkyte@TKYTE816> create table people of person_type
  2  /

Table created.

tkyte@TKYTE816> desc people
 Name             Null?     Type
 ---------------- -------- ---------------------
 NAME                       VARCHAR2(30)
 DOB                        DATE
 HOME_ADDRESS               ADDRESSJTYPE
 WORK ADDRESS               ADDRESS TYPE
```

Вот, собственно, и все. Создаются определения типов, после чего можно создавать таблицы этих типов. Таблица PEOPLE имеет четыре столбца, представляющих четыре атрибута созданного нами типа PERSON_TYPE. Теперь можно применять к объектной таблице операторы ЯМД для создания и выборки данных:

```
tkyte@TKYTE816> insert into people values  ('Tom',  '15-mar-1965',
  2  address_type('Reston',  '123 Main Street', 'Va', '45678'),
  3  address_type('Redwood',  '1 Oracle Way', 'Ca',  '23456'));

1 row created.

tkyte@TKYTE816> select * from people;

NAME   DOB        HOME_ADDRESS(CITY,  S    WORK_ADDRESS (CI

Tom    15-MAR-65  ADDRESS_TYFE('Reston    ADDRESSJTYPE('R
                   ',  '123 Main           edwood',  '1
                   Street',  'Va',          Oracle Way',

                   45678)          'Ca',          23456)

tkyte@TKYTE816> select name,  p.home_address.city  from people p;

NAME   HOME ADDRESS.CITY
------ ------------------------------
Tom    Reston
```

Вам представлены синтаксические конструкции для работы с объектным типами. Например, в операторе INSERT пришлось задавать для столбцов HOME_ADDRESS и WORK_ADDRESS данные с помощью конструкторов. Мы преобразовали набор скалярных значений в тип ADDRESS_TYPE. Экземпляр типа ADDRESS_TYPE для этой строки создан с помощью стандартного конструктора для объекта типа ADDRESS_TYPE.

Итак, в таблице как будто четыре столбца. Но теперь, после выявления скрытых особенностей вложенных таблиц, можно предположить, что это еще не все. Сервер Oracle хранит все объектно-реляционные данные в обычных реляционных таблицах; в конечном итоге все сводится к строкам и столбцам. Если обратиться к "реальному" словарю данных, окажется, что таблица на самом деле выглядит так:

```
tkyte@TKYTE816> select name,  segcollength
  2     from sys.col$
  3    where obj# =  (select object_id
  4                          from user_objects
  5                         where object_name -  'PEOPLE')
  6   /

NAME                  SEGCOLLENGTH
--------------------- ------------
SYS_NC_OID$                     16
SYS_NC_ROWINPO$                  1
NAME                            30
DOB                              7
HOME ADDRESS                     1
```

```
SYS_NC00006$                    30
SYS_NC00007$                    30
SYS_NC00008$                     2
SYS_NC00009$                    22
WORK_ADDRESS                     1
SYS_NC00011$                    30
SYS_NC00012$                    30
SYS_NC00013$                     2
SYS_NC00014$                    22

14 rows selected.
```

Это существенно отличается от того, что выдает команда **describe.** Оказывается, в этой таблице 14 столбцов, а не 4. Они описаны ниже.

- **SYS_NC_OID$.** Это сгенерированный системой идентификатор объекта для таблицы. Это уникальный столбец типа **RAW(16).** Для него установлено ограничение уникальности — по нему создан соответствующий уникальный индекс.
- **SYS_NC_ROWINFO.** Это та же "магическая" функция, что и в случае вложенной таблицы. Если выбрать ее значение из таблицы, возвращается вся строка в виде одного столбца:

```
tkyte@TKYTE816> select sys_nc_rowinfo$ from people;

SYS_NC_ROWINFO$(HAME, DOB, HOME_ADDRESS (CITY, STREET, STATE, ZIP),...

PERSON_TYPE('Tom', 45-MAR-65', ADDRESSJTYPE('Leesburg' , '4234 Main Street',
'Va', 20175), ADDRESS_TYPE('Reston', '4910 Oracle Way', 'Va', 20190))
```

- **NAME, DOB.** Это скалярные атрибуты нашей объектной таблицы. Они, как и можно было предположить, хранятся как обычные столбцы.
- **HOME_ADDRESS, WORK_ADDRESS.** Это тоже "магические" функции — они возвращают набор столбцов в виде единого объекта. Места они не занимают, за исключением признака **NULL** или **NOT NULL** для столбца.
- **SYS_NCnnnnn$.** Это скалярные реализации встроенных объектных типов. Поскольку тип **PERSONJTYPE** содержит встроенный тип **ADDRESS_TYPE,** серверу Oracle необходимо место для хранения его атрибутов в виде столбцов соответствующего типа. Сгенерированные системой имена необходимы, поскольку имена столбцов должны быть уникальными, а ничто не мешает использовать один и тот же объектный тип несколько раз, как в нашем примере. Если бы имена не генерировались, столбец **ZIP,** например, был бы повторен дважды.

Итак, как и в случае вложенных таблиц, "за кулисами" происходит много чего. Добавлен фиктивный первичный ключ длиной 16 байт, автоматически созданы виртуальные столбцы и индекс. Можно изменить стандартный идентификатор объекта, как будет показано ниже. Сначала давайте рассмотри полный текст подробного оператора SQL, генерирующего нашу таблицу. И в этот раз он сгенерирован с помощью пары утилит **EXP/IMP:**

```
CREATE TABLE "TKYTE"."PEOPLE"
OF "PERSON_TYPE" OID '36101E4C6B7E4F7E96A8A6662518965C'
OIDINDEX (PCTFREE 10 INITRANS 2 MAXTRANS 255
          STORAGE(INITIAL 131072 NEXT 131072
                  MINEXTENTS 1 MAXEXTENTS 4096
                  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
                  BUFFER_POOL DEFAULT)
TABLESPACE "USERS")
PCTFREE 10 PCTUSED 40
INITRANS 1 MAXTRANS 255
LOGGING STORAGE(INITIAL 131072 NEXT 131072
                MINEXTENTS 1 MAXEXTENTS 4096
                PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
                BUFFER_POOL DEFAULT) TABLESPACE "USERS"
/

ALTER TABLE "TKYTE"."PEOPLE" MODIFY
("SYS_NC_OID$" DEFAULT SYS_OP_GUID())
/
```

Это дает чуть больше информации о том, что происходит на самом деле. Теперь мы четко видим конструкцию **OIDINDEX** и то, что она ссылается на столбец **SYS_NC_OID$.** Это скрытый первичный ключ таблицы. Функция **SYS_OP_GUID** совпадает с функцией **SYS_GUID.** Обе они возвращают глобально уникальный идентификатор, представленный в поле типа **RAW(16).**

Синтаксис **OID <большое шестнадцатиричное число>'** не описан **в** документации СУБД Oracle. Он просто гарантирует, что в ходе экспорта и последующего импорта базовый тип PERSON_TYPE фактически — **один и тот же.** Это предотвращает возникновение ошибок если:

- создать таблицу **PEOPLE;**
- экспортировать ее;
- удалить базовый тип **PERSON_TYPE;**
- создать новый тип **PERSON_TYPE** с другими атрибутами;
- импортировать старые данные таблицы **PEOPLE.**

Очевидно, экспортированные данные не могут быть импортированы в новую структуру, поскольку они ей не соответствуют. Эта проверка предотвращает возникновение подобной ситуации. Об особенностях экспорта и импорта объектных таблиц см. в главе 8.

Если помните, я упоминал, что можно изменить объектный идентификатор, присваиваемый экземпляру объекта. Вместо автоматически сгенерированного системой фиктивного первичного ключа можно использовать естественный первичный ключ объекта. Сначала это может показаться бесполезным — столбец **SYS_NC_OID$** все равно входит в определение объекта в таблице словаря **SYS.COL$ и,** казалось бы, для него требуется больше пространства, чем для сгенерированного системой столбца. Но и в этом

случае происходит "чудо". Столбец **SYS_NC_OID$** для объектной таблицы, основанный на **первичном ключе,** а не сгенерированный **системой,** является виртуальным и места на диске не занимает. Вот пример, демонстрирующий, что происходит в словаре данных, и то, что физического пространства столбец **SYS_NC_OID$** не занимает. Начнем с анализа таблицы с идентификатором объекта, сгенерированным системой:

```
tkyte@TKYTE816> CREATE  TABLE  "TKYTE"."PEOPLE"
  2   OF  "PERSON_TYPE"
  3   /

Table  created.

tkyte@TKYTE816> select name,  typef,  segcollength
  2     from sys.col$
  3    where  obj# =  (select object_id
  4                          from user_objects
  5                         where  object_name =  ' PEOPLE')
  6      and name like  'SYS\_NC\_%'  escape  '\'
  7   /

NAME                  TYPE#       SEGCOLLENGTH
--------------------- ------  ---------------
SYS_NC_OID$              23                16
SYS_NC_ROWINFO$         121                 1

tkyte@TKYTE816> insert  into  people(name)
  2   select rownum from all_objects;

21765  rows  created.
tkyte@TKYTE816> analyze  table  people  compute  statistics;

Table  analyzed.

tkyte@TKYTE816> select  table_name,  avg_row_len  from  user_object_tables;

TABLE NAME            AVG ROW LEN

PEOPLE                          25
```

Итак, как видите, средняя длина строки составляет 25 байт, 16 байт занимает столбец **SYS_NC_OID$** и 9 байт — столбец **NAME.** Теперь давайте создадим такую же таблицу, но используем первичный ключ, столбец **NAME,** в качестве идентификатора объекта:

```
tkyte@TKYTE816> CREATE TABLE "TKYTE". "PEOPLE"
  2   OF  "PERSON_TYPE"
  3    (constraint people_pk primary key (name))
  4   object  identifier  is  PRIMARY KEY
  5   /

Table  created.

tkyte@TKYTE816> select name,  type#,  segcollength
  2     from sys.col$
  3    where  obj# =  (select object_id
```

```
  4                          from user_objects
  5                         where object name =  'PEOPLE')
  6       and name like  <SYS\_NC\_%'  escape  'V
  7   /

NAME                    TYPE#      SEGCOLLENGTH
---------------------- ------ ----------------
SYS_NC_OID$                 23                81
SYS_NC_ROWINFO$            121                 1
```

Получается, что вместо небольшого столбца размером 16 байт, имеется большой столбец размером 81 байт! На самом деле данные в нем не хранятся. Он — пустой. Система генерирует уникальный идентификатор на основе объектной таблицы, ее базового типа и, конечно, самого значения строки. Это можно продемонстрировать следующим образом:

```
tkyte@TKYTE816> insert into people (name)
  2  values ('Hello World!');

1 row created.

tkyte@TKYTE816> select sys_nc_oid$ from people p;

SYS NC OID$
--------------------------------------------------------------------------------
7129B0A94D3B49258CAC926D8FDD6EEB000000172601000100010029000 0
OOOOOOOC07001E0100002A00078401FE000000140C4B656C6C6F20576F72
6C6421000000000000000000000000000000000000000
```

```
tkyte@TKYTE816> select utl_raw.cast_to_raw('Hello World!') data
  2  from dual;

DATA
--------------------------------------------------------------------------------
48656C6C6F20576F726C6421

tkyte@TKYTE816> select utl_raw.cast_to_varchar2(sys_nc_oid$) data
  2  from people;

DATA
--------------------------------------------------------------------------------
<мусорные данные...>Hello World!
```

Если выбрать данные столбца SYS_NC_OID$ и получить представление вставленной строки в шестнадцатиричном виде, мы увидим, что данные строки встроены в идентификатор объекта. Преобразуя идентификатор объекта в тип VARCHAR2, мы убеждаемся в этом визуально. Значит ли это, что данные хранятся дважды, да еще при этом расходуется большое количество пространства? Нет, конечно.

```
tkyte@TKYTE816> insert into people(name)
  2  select rownum from all_objects;

21766 rows created.

tkyte@TKYTE816> analyze table people compute statistics;
```

```
Table analyzed.

tkyte@TKYTE816> select table_name, avg_row_len from user_object_tables;

TABLE NAME                     AVG ROW LEN
------------------------------ -----------
PEOPLE                                   8
```

Средняя длина строки теперь составляет только 8 байт. На хранение сгенерированного системой ключа пространство больше не тратится, и никакие 81 байт на самом деле не хранятся. Сервер Oracle синтезирует данные этого столбца при выборке из таблицы.

Теперь позволю себе высказать мнение. Объектно-реляционные компоненты (вложенные таблицы, объектные таблицы) я бы назвал "синтаксической приманкой". Они всегда преобразуются в старые, добрые реляционные строки и столбцы. Я лично предпочитаю не использовать их для хранения данных. Слишком много "чудес" происходит, и их побочные эффекты не вполне ясны. Создаются скрытые столбцы, дополнительные индексы, удивительные псевдостолбцы и т.д. **Это не значит, что использование объектно-реляционных компонентов — пустая трата времени** — как раз наоборот. Я постоянно использую их в программах на языке PL/SQL и в объектных представлениях. Я могу получить все преимущества вложенной таблицы (меньший объем передаваемых по сети данных для таблиц, связанных отношениями главный/подчиненные, концептуальная простота использования и т.п.) без сопутствующих проблем физического хранения. И все это благодаря тому, что можно синтезировать объекты из реляционных данных с помощью объектных представлений. Это решает большинство проблем с объектными/вложенными таблицами, поскольку разработчик сам определяет особенности физического хранения и условия соединения, а доступ к базовым таблицам осуществляется, как к обычным реляционным (что требуется для многих инструментальных средств сторонних производителей). Те, кому необходимо объектное представление реляционных данных, его получают, а остальные работают с обычными, реляционными, данными. Поскольку объектные таблицы на самом деле — специальным образом созданные реляционные, мы делаем явно то, что сервер Oracle делает "за кадром", но мы делаем это эффективнее, поскольку не надо давать решение для общего случая. Например, используя определенные ранее типы, я могу использовать следующие простые структуры:

```
tkyte@TKYTE816> create table people_tab
  2   (name          varchar2(30)  primary key,
  3    dob           date,
  4    home_city     varchar2 (30),
  5    home_streat   varchar2(30),
  6    home_state    varchar2(2),
  7    home_zip      number,
  8    work_city     varchar2(30) ,
  9    work_street   varchar2(30),
 10   work_state    varchar2(2),
 11   work_zip      number
 12  )
 13  /
```

```
Table created.

tkyte@TKYTE816> create view people of person_type
  2  with object identifier (name)
  3  as
  4  select name, dob,
  5    address_type(home_city,home_street,home_state,home_zip) home_adress,
  6    address_type(work_city,work_street,work_state,work_zip) work_adress
  7     from people_tab
  8  /

View created.

tkyte@TKYTE816> insert into people values ('Tom', '15-mar-1965',
  2  address_type('Reston', '123 Main Street', 'Va', '45678'),
  3  address_type('Redwood', '1 Oracle Hay', 'Ca', '23456'));

1 row created.
```

Эффект получается почти тот же. Но я точно знаю, что, как и где хранится. Для более сложных объектов, возможно, придется создать триггеры INSTEAD OF для объектных представлений, чтобы можно было изменять данные через представление.

Итак, объектные таблицы используются для реализации объектно-реляционной модели данных в Oracle. Обычно при создании одной объектной таблицы создается много физических объектов базы данных, и в схему добавляются дополнительные столбцы, обеспечивающие управление всеми этими структурами. С объектными таблицами связан ряд "чудес". Объектные представления позволяют воспользоваться синтаксическими и семантическими преимуществами "объектов" и при этом полностью контролировать размещение данных и обеспечивать к ним традиционный, реляционный, доступ. При этом можно получить лучшее из обоих миров — реляционного и объектно-реляционного.

Резюме

Надеюсь, прочитав эту главу, вы поняли, что не все таблицы одинаковы. Сервер Oracle предлагает много различных типов таблиц. В этой главе мы рассмотрели многие скрытые аспекты организации таблиц вообще, а также изучили различные типы таблиц, поддерживаемые сервером Oracle.

Начали мы с изучения терминологии и параметров хранения таблиц. Обсудили назначение списков свободных блоков в многопользовательской среде, где данные часто вставляются и/или изменяются множеством пользователей одновременно. Мы выяснили назначение параметров PCTFREE и PCTUSED, а также выработали ряд принципов их правильной установки.

Затем мы добрались до различных типов таблиц, начиная с обычной, организованной в виде кучи. Таблица, организованная в виде кучи, — стандартный и наиболее широко используемый тип таблиц в большинстве приложений. Затем мы перешли к таблицам, организованным по индексу, которые позволяют хранить данные в индексе. Было показано, как их применять в различных случаях (например, для справочников и

обратных списков, когда организованная в виде кучи таблица просто содержит избыточную копию данных). Затем разобрались, как их использовать в сочетании с другими типами таблиц, в частности, для реализации вложенных таблиц.

Мы рассмотрели два вида кластеров в Oracle — индексные и хеш-кластеры. Создание кластера преследует две цели:

- хранить данные нескольких таблиц вместе, в одних и тех же блоках данных;
- хранить похожие данные в одном месте, на основе ключа кластера, так что, например, все данные для отдела 10 (из нескольких таблиц) хранятся вместе.

Это позволяет получать взаимосвязанные данные очень быстро, минимальным количеством операций ввода/вывода для сбора всех данных. Мы продемонстрировали основное отличие индексных кластеров от хеш-кластеров и обсудили, когда подходит (и не подходит) каждый из типов.

Затем мы перешли к вложенным таблицам. Рассмотрели синтаксис, семантику и использование таких таблиц. Убедились, что фактически они являются сгенерированной и поддерживаемой системой парой таблиц главная/подчиненная, и разобрались, как сервер Oracle создает эти таблицы. Мы рассмотрели различные типы вложенных таблиц, которые по умолчанию организуются в виде кучи. Было показано, что имеет смысл создавать вложенные таблицы как организованные по индексу.

Затем мы рассмотрели особенности временных таблиц: как их создавать, где им выделяется пространство, а также то, что при работе с ними не возникает проблем одновременного доступа. Мы изучили различия между временными таблицами, создаваемыми на время выполнения транзакции и на время сеанса. Был представлен правильный способ использования временных таблиц в базах данных Oracle.

В завершение главы описана работа с объектнымu таблицами. Как и в случае вложенных таблиц, мы обнаружили, что при использовании объектных таблиц в Oracle многое происходит "за кадром". Было показано, как объектные представления на базе обычных реляционных таблиц позволяют получить функциональные возможности объектных таблиц и в то же время обеспечивают простой доступ к базовым реляционным данным. Эта тема более подробно будет рассмотрена в главе 20, посвященной объектно-реляционным возможностям.

7

Индексы

Индексирование — очень важный аспект проектирования и разработки приложения. Если индексов слишком много, снизится производительность операторов ЯМД. Если индексов не хватает, снизится производительность запросов (а следовательно, вставок, изменений и удалений). Правильное решение этой проблемы позволит обеспечить высокую производительность приложений.

Я часто сталкиваюсь с тем, что об индексах при разработке приложений думают в последнюю очередь. Считаю, что это — ошибочный подход. С самого начала, если понятно, как будут использоваться данные, необходимо создать достаточный набор индексов для использования в приложении. Слишком часто приложение просто устанавливают в производственной среде, а потом решают, какие индексы необходимо добавить. Это означает, что разработчики вовремя не оценили способы использования и окончательные объемы данных, с которыми придется работать. Индексы в такой системе придется добавлять бесконечно, по мере роста объемов данных (это называется настройкой постфактум). Появятся избыточные индексы, которые никогда не используются, и на них будет тратиться не только место, но и вычислительные ресурсы. Обдумав в самом начале, когда и как индексировать данные, можно сэкономить большое количество времени, затрачиваемого на настройку по ходу эксплуатации.

В этой главе будет сделан обзор индексов, предлагаемых в СУБД Oracle, и показано, когда и где их можно использовать. Эта глава отличается от других глав книги по стилю и формату. Индексирование — обширная тема; об этом можно написать целую книгу. Индексирование касается как разработчиков приложений, так и администраторов баз данных. Разработчик должен знать об индексах, как применять их в создаваемых приложениях, когда их использовать (а когда — нет) и т.д. Администратор базы данных

должен думать об увеличении индекса, степени его фрагментации и других физических характеристиках. Мы будем рассматривать в основном практическое использование индексов в приложениях (фрагментации индексов и подобным вопросам внимание уделяться не будет). В первой половине этой главы представлены основные сведения, необходимые для принятия обоснованного решения о том, индексировать ли данные и какой тип индекса использовать. Во второй — я дам ответы на некоторые из наиболее часто задаваемых вопросов об индексах.

Различные примеры в этой книге требуют разных базовых версий СУБД Oracle. Если определенная возможность доступна только в Oracle 8i Enterprise или Personal Edition, я это укажу. Многие примеры с индексами на основе B*-дерева требуют использования Oracle 7.0 или более поздних версий. Примеры с битовыми индексами требуют наличия Oracle 7.3.3 или более поздних версий (в редакции Enterprise или Personal). Индексы по функциям и прикладные индексы требуют использования Oracle 8i Enterprise или Personal Edition. Раздел "Часто задаваемые вопросы" применим ко всем версиям СУБД Oracle.

Обзор индексов в Oracle

СУБД Oracle предлагает много различных типов индексов.

- **Индексы на основе B*-дерева.** Эти индексы называют "обычными". Они, несомненно, чаще всего используются в СУБД Oracle, да и в других СУБД. Аналогичные по конструкции двоичному дереву, они обеспечивают быстрый доступ по ключу к отдельной строке или диапазону строк, требуя обычно очень немного чтений для поиска соответствующей строки. Индекс на основе B*-дерева имеет несколько подтипов:

  Таблицы, организованные по индексу. Это таблицы, хранящиеся в структуре B*-дерева. Они достаточно подробно описывались в главе 6, посвященной таблицам. В соответствующем разделе главы 6 рассматривались физические структуры, в которых хранятся B*-деревья, так что к этой теме мы возвращаться не будем.

  Индексы кластера на основе B*-дерева. Они немного отличаются от обычных, используются для индексации ключей кластера (см. соответствующий раздел в главе 6) и отдельно в этой главе рассматриваться не будут. Они используются не для перехода от ключа к строке, а для перехода от ключа кластера к блоку, содержащему строки, связанные с этим ключом.

  Индексы с обращенным ключом. Это индексы на основе B*-дерева, байты ключа в которых инвертированы. Это используется для более равномерного распределения записей по индексу при вводе возрастающих значений ключей. Предположим, при использовании последовательности для генерации первичного ключа генерируются значения 987500, 987501, 987502 и т.д. Поскольку это последовательные значения, они будут попадать в один и тот же блок индекса, конкурируя за него. В индексе с обращенным ключом сервер Oracle будет индексировать значения 205789, 105789, 005789. Эти значения обычно будут далеко отстоять друг от друга в индексе, и вставки в индекс будут распределены по нескольким блокам.

Индексы по убыванию. Далее индексы по убыванию не будут выделяться как отдельный тип. Однако поскольку они только появились в Oracle 8i, то заслуживают отдельного рассмотрения. Индексы по убыванию позволяют отсортировать данные в структуре индекса от "больших" к "меньшим" (по убыванию), а не от меньших к большим (по возрастанию). Мы разберемся, почему это важно и как такие индексы работают.

- **Индексы на основе битовых карт.** Обычно в B*-дереве имеется однозначное соответствие между записью индекса и строкой — запись индекса указывает на строку. В индексе на основе битовых карт запись использует битовую карту для ссылки на большое количество строк одновременно. Такие индексы подходят для данных с небольшим количеством различных значений, которые обычно только читаются. Столбец, имеющий всего три значения — Y, N и NULL, — в таблице с миллионом строк очень хорошо подходит для создания индекса на основе битовых карт. Индексы на основе битовых карт не нужно использовать в базе данных класса OOT из-за возможных проблем с одновременным доступом (которые мы рассмотрим далее).

- **Индексы по функции.** Эти индексы на основе B*-дерева или битовых карт хранят вычисленный результат применения функции к столбцу или столбцам строки, а не сами данные строки. Это можно использовать для ускорения выполнения запросов вида: SELECT * **FROM T WHERE ФУНКЦИЯ(СТОЛБЕЦ) = НЕКОТОРОЕ_ЗНАЧЕНИЕ,** поскольку значение **ФУНКЦИЯ(СТОЛБЕЦ)** уже вычислено и хранится в индексе.

- **Прикладные (application domain) индексы.** Это индексы, которые строит и хранит приложение, будь-то в базе данных Oracle или даже вне базы данных Oracle. Надо сообщить оптимизатору, насколько избирателен индекс, насколько "дорогостояще" его использование, а оптимизатор решает на основе этой информации, использовать этот индекс или нет. Текстовый индекс **interMedia** — пример прикладного индекса; он построен с помощью тех же средств, которые можно использовать для создания собственных прикладных индексов.

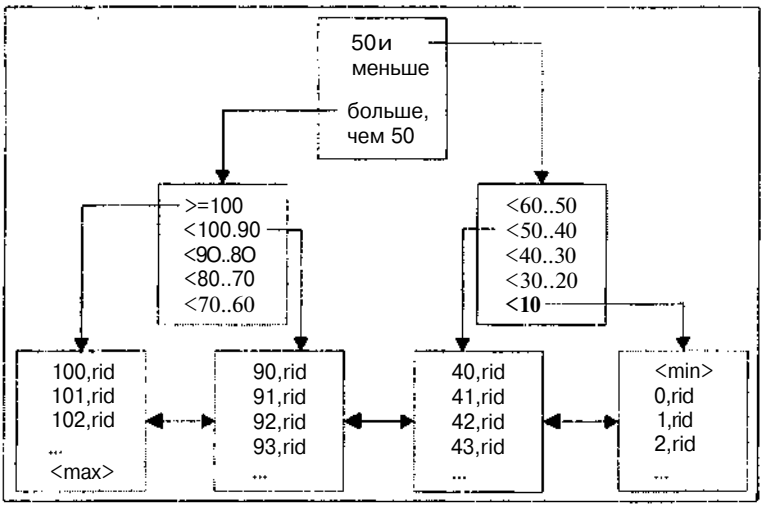
- **Текстовые индексы interMedia.** Это встроенные в сервер Oracle специализированные индексы для обеспечения поиска ключевых слов в текстах большого объема. Описание этих индексов будет представлено в главе 17, посвященной компоненту **interMedia.**

Как видите, предлагается несколько типов индексов на выбор. В следующих разделах я хочу представить технические детали их работы и порекомендовать, когда их использовать. Еще раз подчеркну: мы не будем рассматривать ряд вопросов, интересующих администраторов баз данных (например, механизм оперативной перестройки индекса), а сосредоточимся на практическом использовании индексов в приложениях.

Индексы на основе B*-дерева

Индексы на основе B*-дерева, или, как я их назвал, "обычные" индексы, — наиболее широко используемый тип индексной структуры в базе данных. По реализации они

подобны двоичному дереву поиска. Цель их создания — минимизировать время поиска данных сервером Oracle. При наличии индекса по числовому столбцу, структура индекса может выглядеть так:



Блоки самого нижнего уровня в индексе, которые называют *листовыми вершинами,* содержат все проиндексированные ключи и идентификаторы строк (rid на схеме), ссылающиеся на соответствующие строки. Промежуточные блоки над листовыми вершинами называют блоками ветвления. Они используются для переходов по структуре. Например, если необходимо найти в индексе значение 42, надо начать с вершины дерева и двигаться вправо. При проверке этого блока оказывается, что необходимо перейти к блоку в диапазоне "от 40 до 50". Этот блок оказывается листовым и ссылается на строки, содержащие число 42. Интересно отметить, что листовые блоки фактически образуют двухсвязный список. Как только найдено "начало" среди листовых вершин, т.е. первое значение, очень легко просматривать значения по порядку (это называют также *просмотром диапазона по индексу,* index range scan). Проходить по структуре индекса больше не нужно; мы просто переходим по листовым вершинам. Это существенно упрощает поиск строк но условиям следующего вида:

```
where x between 20 and 30
```

Сервер Oracle находит первый блок индекса, содержащий значение 20, а затем проходит по двухсвязному списку листовых вершин, пока не обнаружит значение больше 30.

На самом деле такой структуры, как неуникальный индекс на основе B*-дерева, нет. В неуникальном индексе сервер Oracle просто добавляет идентификаторы строк к ключу индекса, что и делает его неуникальным. В неуникальном индексе данные хранятся отсортированными сначала по значению ключа индекса (в порядке, задаваемом ключом индекса), а потом — по идентификатору строки. В уникальном индексе данные отсортированы только по значению ключа индекса.

Одно из свойств B*-дерева состоит в том, что все листовые блоки должны быть на одном уровне, если точнее, разница по высоте между ветвями дерева не может быть

больше 1. Уровень листовых блоков называют также *высотой дерева*. Все вершины выше листовых могут указывать только на содержащие более детальную информацию вершины следующего уровня, а листовые вершины указывают на конкретные идентификаторы строк или диапазоны идентификаторов строк. Большинство индексов на основе B*-дерева будут иметь высоту 2 или 3, даже для миллионов записей. Это означает, что для поиска ключа в индексе потребуется 2 или 3 чтения, что неплохо. Еще одно свойство — автоматическая балансировка листовых вершин: они почти всегда располагаются на одном уровне. Есть причины, по которым индекс может оказаться не идеально сбалансированным при изменении и удалении записей. Сервер Oracle будет пытаться заполнять блоки индекса не более чем на три четверти, но и это свойство может временно нарушаться при выполнении операторов DELETE и UPDATE. В общем случае B*-дерево — отличный универсальный механизм индексирования, хорошо работающий как в случае больших, так и маленьких таблиц, и лишь немного хуже работающий при росте базовой таблицы, если только дерево остается сбалансированным.

Интересно, что индексы на основе B*-дерева можно "сжимать". Это не такое сжатие, как в zip-файлах; при сжатии удаляется избыточность в составных индексах. Мы уже рассматривали этот механизм в разделе "Таблицы, организованные по индексу" главы 6, но вернемся к нему еще раз. В основе сжатия ключа индекса лежит разбиение записи на две части: префикс и суффикс. Префикс строится по начальным столбцам составного индекса, и его значения часто повторяются. Суффикс — это завершающие столбцы ключа индекса, и эта часть в записях индекса с одинаковым префиксом — уникальна. Давайте создадим таблицу и индекс, и определим используемое им пространство без сжатия, а затем пересоздадим индекс с включенным сжатием, и оценим разницу.

В этом примере используется процедура show_space, представленная в главе 6.

```
tkyte@TKYTE816> create table t
  2  as
  3  select * from all_objects
  4  /

Table created.

tkyte@TKYTE816> create index t_idx on
  2  t(owner,object_type,object_name);

Index created.

tkyte@TKYTE816>
tkyte@TKYTE816> exec show_space('T_IDX',user,'INDEX')
Free Blocks.....0
Total Blocks.....192
Total Bytes.....1572864
Unused Blocks.....35
Unused Bytes.....286720
Last Used Ext FileId.....6
Last Used Ext BlockId.....649
Last Used Block.....29

PL/SQL procedure successfully completed.
```

Для индекса выделено 192 блока, 35 из которых не содержат данных (всего используется 157 блоков). Понятно, что компонент OWNER повторяется многократно. Один блок индекса будет включать десятки записей вида:

```
Sys,Package,Dbms_Alert
Sys,Package,Dbms_Application_Info
Sys,Package,Dbms_Aq
Sys,Package,Dbms_Aqadm
Sys,Package,Dbms_Aqadm_Sys
Sys,Package,Dbms_Aqadm_Syscalls
Sys,Package,Dbms_Aqin
Sys,Package,Dbms_Aqjms
....
```

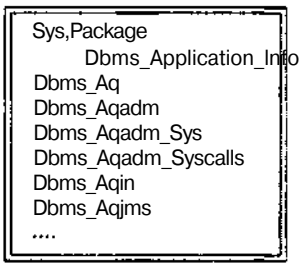
Можно вынести *(факторизовать)* один повторяющийся столбец, OWNER, и получить в результате блок, выглядящий примерно так:

```
Sys
Package,Dbms_Alert
Package,Dbms_Application_Info
Package,Dbms_Aq
Package,Dbms_Aqadm
Package,Dbms_Aqadm_Sys
Package,Dbms_Aqadm_Syscalls
Package,Dbms_Aqin
Package,Dbms_Aqjms
....
```

Здесь имя владельца появляется в листовом блоке только один раз, а не для каждой записи. Если пересоздать индекс со сжатием первого столбца:

```
tkyte@TKYTE816> drop index t_idx;
Index dropped.

tkyte@TKYTE816> create index t_idx on
  2   t(owner,object_type,object_name)
  3  compress 1;
Index created.

tkyte@TKYTE816> exec show_space('T_IDX',user,'INDEX')
Free Blocks.....0
Total Blocks.....192
Total Bytes.....1572864
Unused Blocks.....52
Unused Bytes.....425984
Last Used Ext FileId.....6
Last Used Ext BlockId.....649
Last Used Block.....12

PL/SQL procedure successfully completed.
```

Размер индексной структуры уменьшается со 157 блоков до 140, примерно на 10 процентов. Можно продолжить эксперимент и сжать два первых столбца. Это приведет к созданию блоков, в которых вынесены повторяющиеся элементы столбцов OWNER и OBJECT TYPE:



Теперь, после сжатия первых двух столбцов:

```
tkyte@TKYTE816> drop index t_idx;
Index dropped.

tkyte@TKYTE816> create index t_idx on
  2    t(owner,object_type,object_name)
  3    compress 2;
Index created.

tkyte@TKYTE816>
tkyte@TKYTE816> exec show_space('T_IDX',user,'INDEX')
Free Blocks.....0
Total Blocks.....128
Total Bytes.....1048576
Unused Blocks.....15
Unused Bytes.....122880
Last Used Ext FileId.....6
Last Used Ext BlockId.....585
Last Used Block.....49

PL/SQL procedure successfully completed.
```

мы получили индекс из 113 блоков, почти на тридцать процентов меньше исходного. В зависимости от повторяемости данных, экономия может оказаться и более существенной. Но это сжатие не дается даром. Структура сжатого индекса усложняется. Сервер Oracle будет тратить больше времени на обработку данных в этой структуре как при поддержке индекса в ходе изменения, так и при поиске в ходе выполнения запроса. Мы пошли на увеличение процессорного времени обработки при одновременном уменьшении времени на выполнение ввода/вывода. В буферный кэш поместится больше записей индекса, чем при отсутствии сжатия, процент попадания в буферный кэш может увеличиться, объем физического ввода/вывода — уменьшится, но на обработку индекса потребуется больше времени, да и вероятность конфликтов при доступе к блоку возрастает. Как и в случае хеш-кластера, когда для извлечения миллиона случайных строк потребовалось больше процессорного времени, но в два раза меньше операций ввода/вывода, необходимо помнить об этом компромиссе. Если вычислительная мощность

ограничена, добавление индексов со сжатым ключом может замедлить работу. С другой стороны, если главным требованием является скорость выполнения операций ввода/вывода, их использование позволит ускорить работу.

Индексы с обращенным ключом

Еще одна особенность индексов на основе B*-дерева — возможность "обратить" ключи. Сначала может показаться странным, зачем это вообще нужно? Они созданы для специфической среды и с конкретной целью. Они предназначены для уменьшения количества конфликтов при доступе к листовым блокам индекса в среде Oracle Parallel Server (OPS).

Конфигурация OPS описана в главе 2, посвященной архитектуре сервера Oracle.

В этой конфигурации сервера Oracle несколько экземпляров могут монтировать и открывать одну и ту же базу данных. Если двум экземплярам одновременно необходимо изменить один и тот же блок данных, они совместно используют этот блок, сбрасывая его на диск, чтобы другой экземпляр мог его прочитать. Это действие называют тестовым опросом (pinging). Тестовых опросов при использовании конфигурации OPS надо избегать, но они практически неизбежны при использовании обычного индекса на основе B*-дерева по столбцу, значения которого генерируются с помощью последовательности. Все экземпляры будут пытаться изменить левый конец индексной структуры при вставке новых значений (см. схему в начале раздела "Индексы на основе B*-дерева", показывающую, что "большие значения" в индексе попадают налево, а меньшие — направо). В среде OPS изменения индексов по столбцам, заполняемым последовательными значениями, сосредоточены на небольшом подмножестве листовых блоков. Обращение ключей индекса позволяет распределить вставки по всем листовым блокам индекса, хотя в результате индекс обычно менее плотно упакован.

В индексе с обращенным ключом просто обращается порядок байтов в каждом столбце ключа индекса. Если взять числа 90101, 90102, 90103 и посмотреть на их внутреннее представление с помощью встроенной функции DUMP, окажется, что они представлены следующим образом:

```
tkyte@TKYTE816> select 90101, dump(90101,16)  from dual
  2  union all
  3  select 90102, dump(90102,16)  from dual
  4  union all
  5  select 90103, dump(90103,16)  from dual
  6  /

 90101 DUMP(90101,16)

 90101 Typ=2 Len=4: c3,a,2,2
 90102 Typ=2 Len=4: c3,a,2,3
 90103 Typ=2 Len=4: c3,a,2,4
```

Каждое число представлено четырьмя байтами, и отличаются они только последним байтом. В структуре индекса эти числа окажутся рядом. Если же обратить порядок следования байтов, сервер Oracle вставит следующие значения:

```
tkyte@TKYTE816> select 90101, dump(reverse(90101),16) from dual
  2  union all
  3  select 90102, dump(reverse(90102),16) from dual
  4  union all
  5  select 90103, dump(reverse(90103),16) from dual
  6  /

 90101 DUMP(REVERSE(90101),1
------ --------------------
 90101 Typ=2 Len=4: 2,2,a,c3
 90102 Typ=2 Len=4: 3,2,a,c3
 90103 Typ=2 Len=4: 4,2,a,c3
```

Эти числа окажутся "далеко" друг от друга. При этом сокращается количество экземпляров, обращающихся к одному и тому же блоку (крайнему слева) и, следовательно, количество выполняемых тестовых опросов. Один из недостатков индекса с обращенными ключами — то, что его нельзя использовать в некоторых случаях, когда обычный индекс вполне применим. Например, при поиске по следующему критерию индекс с обращенным ключом по столбцу x не поможет:

```
where x > 5
```

Данные в индексе не отсортированы, поэтому просмотреть диапазон нельзя. С другой стороны, некоторые просмотры диапазонов в индексе с обращенным ключом вполне выполнимы. Если имеется составной индекс по столбцам X, Y, при поиске по следующему условию можно будет использовать индекс с обращенным ключом и "просматривать диапазон" в нем:

```
where x = 5
```

Дело в том, что байты в столбце X обращены, и байты в столбце Y тоже обращены. Сервер Oracle не обращает байты значения X || Y, а сохраняет в записи индекса результат выполнения reverse(X) || reverse(Y). Это означает, что все значения X = 5 будут храниться вместе, так что сервер Oracle может просматривать последовательно листовые блоки индекса для поиска всех таких строк.

Индексы по убыванию

Индексы по убыванию — новое средство сервера Oracle 8i, расширяющее функциональные возможности индекса на основе B*-дерева. Они позволяют хранить значения столбца в индексе от "большего" к "меньшему", а не по возрастанию. Прежние версии сервера Oracle всегда поддерживали ключевое слово DESC (по убыванию), но при этом игнорировали его — оно не влияло на хранение и использование данных в индексе. В версии Oracle 8i, однако, это ключевое слово изменяет способ создания и использования индексов.

Сервер Oracle давно может просматривать индексы в обратном порядке, поэтому кажется странным, зачем такая возможность вообще понадобилась. Например, если использовать таблицу T из предыдущего примера и выполнить следующий запрос:

```
tkyte@TKYTE816> select owner, object_type
  2  from t
```

```
  3  where owner between 'T' and 'Z'
  4   and object_type is not null
  5   order by owner DESC, object_type DESC
  6   /
46 rows selected.


Execution Plan
----------------------------------------------------------------------------
   0       SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=46 Bytes=644)
   1    0   INDEX (RANGE SCAN DESCENDING) OF 'T_IDX' (NON-UNIQUE) . . .
```

Оказывается, что сервер Oracle будет просто читать индекс в обратном порядке, поскольку в этом плане выполнения нет завершающей сортировки — данные и так отсортированы. Возможность создавать индекс по убыванию имеет значение только для составного индекса, в котором некоторые столбцы упорядочены по возрастанию (ASC), а некоторые — по убыванию (DESC). Например:

```
tkyte@TKYTE816> select owner, object_type
  2   from t
  3  where owner between 'T' and 'Z'
  4   and object_type is not null
  5   order by owner DESC, object_type ASC
  6   /
46 rows selected.


Execution Plan

   0       SELECT STATEMENT Optimizer=CHOOSE (Cost=4 Card=46 Bytes=644)
   1    0    SORT (ORDER BY) (Cost=4 Card=46 Bytes=644)
   2    1      INDEX (RANGE SCAN) OF 'T_IDX' (NON-UNIQUE) (Cost=2 Card=
```

Сервер Oracle больше не может использовать имеющийся индекс по столбцам (OWNER, OBJECT_TYPE, OBJECT_NAME) для сортировки данных. Он мог бы читать его в обратном порядке для получения данных, отсортированных по критерию OWNER DESC, но ему надо читать их по возрастанию, чтобы получить отсортированные по возрастанию данные в столбце OBJECT_TYPE. Поэтому по индексу выбираются все строки, а затем сортируются. Здесь поможет индекс с ключевым словом DESC:

```
tkyte@TKYTE816> create index desc_t_idx on t(owner DESC, object_type ASC)
  2  /
Index created.

tkyte@TKYTE816> select owner, object_type
  2   from t
  3  where owner between 'T' and 'Z'
  4   and object_type is not null
  5   order by owner DESC, object_type ASC
  6   /
46 rows selected.
```

```
Execution Plan
----------------------------------------------------------------------
```

Индексы 351

```
Execution Plan
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=4 Card=46 Bytes=644)
   1    0   INDEX (RANGE SCAN) OF 'DESC_T_IDX' (NON-UNIQUE)...
```

Теперь опять можно читать отсортированные данные — дополнительного шага сортировки в конце плана нет. Учтите, что если параметр **COMPATIBLE** в файле **init.ora** не имеет значения **8.1.0** или выше, опция DESC в операторе **CREATE INDEX** будет проигнорирована: никаких предупреждений или сообщений об ошибке выдано не будет, поскольку это — стандартное поведение для прежних версий сервера.

Когда имеет смысл использовать индекс на основе B*-дерева?

Не слишком веря в "простые" правила (из каждого правила есть исключения), я не использую никаких простых правил для определения того, когда использовать (или не использовать) индекс на основе B*-дерева. Чтобы обосновать свою точку зрения, я представлю два одинаково верных правила:

- используйте индексы на основе B*-дерева по столбцу, если предполагается выбирать из таблицы по индексу лишь небольшую часть строк;
- используйте индекс на основе B*-дерева, если предполагается обработка множества строк таблицы и можно использовать индекс **вместо** таблицы.

Эти правила, казалось бы, противоречат друг другу, но на самом деле это не так — просто они предназначены для двух принципиально разных случаев. Есть два способа использовать индекс.

- **Как средство доступа к строкам в таблице.** Индекс читается, чтобы добраться до строки в таблице. Так имеет смысл обращаться к очень небольшой части строк таблицы.
- **Как средство ответа на запрос.** Индекс содержит достаточно информации, чтобы дать полный ответ на запрос — к таблице вообще не придется обращаться. Индекс будет использоваться как уменьшенная версия таблицы.

Первое правило относится к случаю, когда имеется таблица T (используем таблицу T из предыдущего примера) и применяется следующий план выполнения запроса:

```
tkyte@TKYTE816> set autotrace traceonly explain

tkyte@TKYTE816> select owner, status
  2     from T
  3    where owner = USER;

Execution Plan

   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0    TABLE ACCESS (BY INDEX ROWID) OF 'T'
   2    1      INDEX (RANGE SCAN) OF 'T_IDX' (NON-UNIQUE)
```

Так можно обращаться к небольшой части этой таблицы. Надо обращать внимание на шаг **INDEX (RANGE SCAN),** после которого идет **TABLE ACCESS BY INDEX ROWID.** Это означает, что сервер Oracle будет читать индекс, а затем для каждой записи индекса будет читать блок (логически или физически) с данными строки. Это — не самый эффективный метод, если предполагается доступ по индексу к большому количеству строк таблицы T (ниже мы определим, какого размера может быть эта существенная часть).

С другой стороны, если можно использовать индекс **вместо** таблицы, по индексу можно обрабатывать хоть все строки **таблицы** (или любую часть). Об этом и говорит второе простое правило. Можно использовать индекс как "облегченную" версию таблицы (с упорядоченными строками).

Следующий запрос демонстрирует эту идею:

```
tkyte@TKYTE816> select count(*)
  2     from T
  3    where owner = USER;

Execution Plan
----------------------------------------------------------
    0       SELECT STATEMENT Optimizer=CHOOSE
    1    0     SORT (AGGREGATE)
    2    1       INDEX (RANGE SCAN) OF 'T_IDX' (NON-UNIQUE)
```

Здесь для выполнения запроса использовался только индекс — неважно, к какому количеству строк мы обращались, ведь использовался только индекс. По плану выполнения понятно, что к базовой таблице вообще не обращались — просматривалась только структура индекса.

Важно понять различие между этими двумя случаями. Если необходимо выполнить **TABLE ACCESS BY INDEX ROWID,** надо гарантировать, что так мы обращаемся к небольшой части строк таблицы. При таком способе доступа к слишком большому количеству строк (предел — от 1 до 20 процентов строк) он будет выполняться дольше, чем полный просмотр таблицы. Для запросов второго типа, ответ на которые целиком находится в индексе, ситуация принципиально отличается. Мы читаем блок индекса и выбираем множество строк для обработки, затем, переходим к следующему блоку индекса и т.д., вообще не обращаясь к таблице. По индексам можно также делать быстрый полный просмотр, т.е. благодаря им в определенных случаях работа выполняется еще быстрее. Быстрый полный просмотр — это когда сервер читает блоки индекса не в определенном порядке — он их просто читает. Индекс уже не используется как индекс, скорее — как таблица. При быстром полном просмотре записей индекса строки выдаются неупорядоченными.

Обычно индекс на основе B*-дерева я создаю по столбцам, часто используемым в условии запроса, если предполагается, что по запросу будет возвращаться небольшая часть строк. В простенькой таблице с небольшим количеством или размером столбцов эта часть может быть очень маленькой. Запрос, использующий индекс, обычно должен извлекать не более 2—3 процентов строк таблицы. В массивной таблице с большим количеством или размером столбцов эта доля **может** доходить до 20-25 процентов таблицы. Этот совет не каждый сразу воспримет; он интуитивно не понятен, но правилен.

Индекс хранится отсортированным по ключу. Индекс будет просматриваться в порядке сортировки ключей. Блоки, на которые указывает индекс, хранятся в случайном порядке, в виде кучи. Поэтому при доступе к таблице по индексу придется выполнять множество разрозненных, случайных операций ввода/вывода. Разрозненность объясняется тем, что по индексу придется читать блок 1, блок 1000, блок 205, блок 321, блок 1, блок 1032, блок 1 и т.д., а не последовательно блок 1, затем 2, 3 и т.д. Придется читать блоки случайным образом. В этом случае ввод/вывод одного блока может выполняться крайне медленно.

В качестве упрощенного примера возьмем несложную таблицу, читаемую по индексу, при условии, что должно быть прочитано 20 процентов строк. Предположим, в таблице 100000 строк. Двадцать процентов от этого составляет 20000 строк. Если строки в среднем имеют длину 80 байт, в базе данных с размером блока 8 Кбайт в нем будет помещаться около 100 строк. Это означает, что в таблице — около 1000 блоков. Теперь рассчитать все будет несложно. Мы собираемся прочитать по индексу 20000 строк, другими словами, выполнить 20000 операций TABLE ACCESS BY ROWID. Для выполнения этого запроса придется обработать 20000 блоков таблицы. Хотя во всей таблице всего лишь около 1000 блоков! В конечном итоге окажется, что мы прочитали и обработали каждый блок в таблице в среднем 20 раз! Даже если увеличить размер строки на порядок (до 800 байт), что дает 10 строк в блоке, — такая таблица займет 10000 блоков. При доступе по индексу к 20000 строк нам придется прочитать каждый блок в среднем дважды. В данном случае полный просмотр таблицы будет намного эффективнее, чем доступ по индексу, поскольку каждый блок будет просматриваться только один раз. Запрос, использующий этот индекс для доступа к данным, не будет выполняться эффективно, если в среднем обращается более чем к 5 процентам данных для столбца размером 800 байт (в этом случае мы обратимся примерно к 5000 блоков) или к еще меньшей части (менее 0,5 процента) для столбца размером 80 байт.

Разумеется, есть факторы, влияющие на эти расчеты. Предположим, имеется таблица с первичным ключом, заполняемым на основе последовательности. При добавлении данных к таблице строки с последовательными значениями первичного ключа обычно будут располагаться "рядом". Таблица естественным образом кластеризуется, упорядочиваясь по первичному ключу (поскольку именно в таком порядке данные добавляются). Она, конечно же, не будет строго упорядоченной по ключу (чтобы добиться этого, надо использовать таблицу, организованную по индексу), но в большинстве случаев строки с первичными ключами, имеющими близкие значения, физически располагаются достаточно "близко". Теперь, если выполнить запрос:

```
select * from T where primary_key between :x and :y
```

необходимые строки будут находиться в одних и тех же блоках. В этом случае сканирование диапазона по индексу может пригодиться, даже если при этом обращаются к существенной части строк, просто потому, что блоки базы данных, которые мы будем читать и перечитывать, скорее всего будут находиться в кэше, поскольку данные размещены рядом. С другой стороны, если строки размещены вразброс, использование того же индекса может дать катастрофически низкую производительность. Продемонстрируем это на небольшом примере. Мы начнем с таблицы, практически упорядоченной по первичному ключу:

```
tkyte@TKYTE816> create table colocated (x int, y varchar2(2000)) pctfree 0;
Table created.

tkyte@TKYTE816> begin
  2    for i in 1 .. 100000
  3     loop
  4              insert into colocated values (i,
rpad(dbms_random.random, 75, '*'));
  5     end loop;
  6  end;
  7  /
PL/SQL procedure successfully completed.

tkyte@TKYTE816> alter table colocated
  2  add constraint colocated_pk primary key(x);
Table altered.
```

Эта таблица соответствует приведенному выше описанию: около 100 строк в блоке при размере блока 8 Кбайт. Весьма вероятно, что в этой таблице строки со значением x = 1, 2, 3 попадут в один блок. Теперь мы умышленно дезорганизуем эту таблицу. В представленной выше таблице COLOCATED мы создали столбец Y со случайными значениями, и теперь используем его для дезорганизации данных, так что они уж точно не будут больше упорядочены по первичному ключу:

```
tkyte@TKYTE816> create table disorganized nologging pctfree 0
  2  as
  3  select x, y from colocated ORDER BY y
  4  /
Table created.

tkyte@TKYTE816> alter table disorganized
  2 add constraint disorganized_pk primary key(x);
Table altered.
```

Можно утверждать, что это — одинаковые таблицы. Это же реляционная СУБД, и физическая организация не влияет на результаты (по крайней мере, так учат в теоретических курсах по базам данных). На самом деле характеристики производительности этих двух таблиц отличаются, как небо и земля. Выполняем один и тот же запрос:

```
tkyte@TKYTE816> select * from COLOCATED where x between 20000 and 40000;
20001 rows selected.
Elapsed: 00:00:01.02

Execution Plan
----------------------------------------------------------------------
    0      SELECT STATEMENT Optimizer=CHOOSE
    1    0   TABLE ACCESS (BY INDEX ROWID) OF 'COLOCATED'
    2    1     INDEX (RANGE SCAN) OF 'COLOCATED_PK' (UNIQUE)

Statistics
----------------------------------------------------------
          0 recursive calls
          0 db block gets
```

```
      2909  consistent gets
       258  physical reads
         0  redo size
   1991367  bytes sent via SQL*Net to client
    148387  bytes received via SQL*Net from client
      1335  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
     20001  rows processed

tkyte@TKYTE816> select * from DISORGANIZED where x between 20000 and 40000;
20001 rows selected.
Elapsed: 00:00:23.34

Execution Plan

    0       SELECT STATEMENT Optimizer=CHOOSE
    1    0    TABLE ACCESS (BY INDEX ROWID) OF 'DISORGANIZED'
    2    1      INDEX (RANGE SCAN) OF 'DISORGANIZED_PK' (UNIQUE)

Statistics

         0  recursive calls
         0  db block gets
     21361  consistent gets
      1684  physical reads
         0  redo size
   1991367  bytes sent via SQL*Net to client
    148387  bytes received via SQL*Net from client
      1335  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
     20001  rows processed
```

Это просто невероятно — насколько существенно может влиять на результат физическое размещение данных! Подведем итоги:

Таблица	*Время выполнения*	*Логические операции ввода/вывода*
Colocated	1,02 секунды	2909
Disorganized	23,34 секунды	21361

В моей базе данных с размером блока 8 Кбайт каждая из этих таблиц занимает 1088 блоков. Запрос к дезорганизованной таблице демонстрирует рассчитанный ранее результат: выполнено более 20000 логических операций ввода/вывода. Каждый блок мы обрабатывали 20 раз! С другой стороны, при физическом размещении близких данных рядом количество логических операций заметно уменьшается. Вот отличный пример того, почему простые правила так сложно сформулировать: в одном случае использование индекса эффективно, в других — нет. Учтите это в следующий раз, когда будете сбрасывать данные из производственной системы и переносить в среду разработки — это

поможет ответить на часто возникающий вопрос: "Почему на этой машине работает не так — они же идентичны?". Они не идентичны.

Чтобы завершить этот пример, давайте посмотрим, что происходит при полном просмотре дезорганизованной таблицы:

```
tkyte@TKYTE816> select /*+ FULL(DISORGANIZED) */ *
  2   from DISORGANIZED
  3   where x between 20000 and 40000;

20001 rows selected.

Elapsed: 00:00:01.42

Execution Plan

    0      SELECT STATEMENT Optimizer=CHOOSE (Cost=162 Card=218 Bytes-2
    1    0   TABLE ACCESS (FULL) OF 'DISORGANIZED' (Cost=162 Card=218 B

Statistics

          0  recursive calls
         15  db block gets
       2385  consistent gets
        404  physical reads
          0  redo size
    1991367  bytes sent via SQL*Net to client
     148387  bytes received via SQL*Net from client
       1335  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      20001  rows processed
```

Это показывает, что в данном случае способ физического хранения данных на диске вполне допускает полный просмотр. Возникает вопрос: как это учесть? Ответ простой: используйте оптимизатор, основанный на стоимостей он сделает это автоматически. Представленный выше пример выполнялся в режиме оптимизации RULE, поскольку статистическая информация о таблице не собиралась. Единственный раз, когда использовался оптимизатор, основанный на стоимости, — при вводе подсказки о полном просмотре таблицы, — мы попросили его сгенерировать конкретный план. Проанализировав таблицы, можно увидеть часть информации, используемой сервером Oracle для оптимизации представленных выше запросов:

```
tkyte@TKYTE816> analyze table colocated
  2   compute statistics
  3   for table
  4   for all indexes
  5   for all indexed columns
  6   /
Table analyzed.

tkyte@TKYTE816> analyze table disorganized
  2   compute statistics
```

```
  3  for table
  4  for all indexes
  5  for all indexed columns
  6  /
Table analyzed.
```

Теперь давайте получим часть используемой сервером Oracle информации. В частности, нас будет интересовать значение столбца **CLUSTERING_FACTOR** в представлении **USER_INDEXES**. Руководство *Oracle Reference Manual* утверждает, что этот столбец имеет следующий смысл.

Показывает, насколько упорядочены строки в таблице по значениям индекса:

- Если значение близко к общему количеству блоков, значит, таблица очень хорошо упорядочена. В этом случае записи индекса в одном листовом блоке обычно указывают на строки, находящиеся в одних и тех же блоках данных.
- Если значение близко к общему количеству строк, значит, таблица весьма неупорядочена. В этом случае маловероятно, что записи индекса в одном листовом блоке указывают на те же блоки данных.

Значение **CLUSTERING_FACTOR** (показатель кластеризации) — показатель того, насколько упорядочена таблица в соответствии с индексом. Посмотрим на информацию об использовавшихся нами индексах:

```
tkyte@TKYTE816> select a.index_name,
  2               b.num_rows,
  3               b.blocks,
  4               a.clustering_factor
  5    from user_indexes a, user_tables b
  6  where index_name in ('COLOCATED_PK', 'DISORGANIZED_PK')
  7    and a.table_name = b.table_name
  8  /

INDEX_NAME           NUM_ROWS          BLOCKS     CLUSTERING_FACTOR

COLOCATED_PK           100000            1063                  1063
DISORGANIZED_PK        100000            1064                 99908
```

Индекс COLOCATED_PK — классический пример "хорошо упорядоченной таблицы", а вот DISORGANIZE_PK — как раз классический пример "неупорядоченной таблицы". Интересно разобраться, как это влияет на работу оптимизатора. Если попытаться выбрать 20000 строк, сервер Oracle теперь использует полный просмотр таблицы для обоих запросов (выбор 20 процентов строк по индексу — неоптимальный план даже для очень хорошо упорядоченной таблицы). Однако если выбирать 10 процентов данных таблицы:

```
tkyte@TKYTE816> select * from COLOCATED where x between 20000 and 30000;
10001 rows selected.

Elapsed: 00:00:00.11
```

```
Execution Plan

    0      SELECT STATEMENT Optimizer=CHOOSE (Cost=129 Card=9996 Bytes=839664)
    1   0   TABLE ACCESS (BY INDEX ROWID) OF 'COLOCATED' (Cost=129 Card=9996
    2   1     INDEX (RANGE SCAN) OF 'COLOCATED_PK' (UNIQUE) (Cost=22 Card=9996)

Statistics

          0  recursive calls
          0  db block gets
       1478  consistent gets
        107  physical reads
          0  redo size
     996087  bytes sent via SQL*Net to client
      74350  bytes received via SQL*Net from client
        668  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
      10001  rows processed

tkyte@TKYTE816> select * from DISORGANIZED where x between 20000 and 30000;
10001 rows selected.

Elapsed: 00:00:00.42

Execution Plan

    0      SELECT STATEMENT Optimizer=CHOOSE (Cost=162 Card=9996 Bytes=839664)
    1   0   TABLE ACCESS (FULL) OF 'DISORGANIZED' (Cost=162 Card=9996

Statistics

          0  recursive calls
         15  db block gets
       1725  consistent gets
        707  physical reads
          0  redo size
     996087  bytes sent via SQL*Net to client
      74350  bytes received via SQL*Net from client
        668  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
      10001  rows processed
```

Мы получаем те же структуры таблиц, те же индексы, но разные показатели кластеризации. Оптимизатор в этом случае выбрал доступ по индексу для таблицы **COLOCATED** и полный просмотр — для таблицы **DISORGANIZED.**

Итак, доступ по индексу — не всегда лучший способ доступа. Оптимизатор может вполне обоснованно отказаться от использования индекса, как продемонстрировано в предыдущем примере. На использование индекса оптимизатором влияет много факторов, **в том числе** физическая организация данных. С учетом этого, можно прямо сейчас попытаться перестроить все таблицы, чтобы все индексы имели хороший показатель кла-

стеризации, но в большинстве случаев это будет напрасной потерей времени. Это повлияет только на просмотры диапазонов по индексу, затрагивающие значительную часть строк таблицы, — не такое уж типичное действие, по моему опыту. Кроме того, надо помнить, что обычно таблица имеет только **один** индекс с хорошим показателем кластеризации! Данные могут быть отсортированы только по одному критерию. В предыдущем примере, если бы был еще один индекс по столбцу Y, он оказался бы очень слабо кластеризованным в таблице **DISORGANIZED.** Если физическое упорядочение данных существенно, пересоздайте таблицу, организовав ее по соответствующему индексу.

Итак, индексы на основе B*-дерева, несомненно, — самые типичные и понятные индексные структуры в базе данных Oracle. Это отличный универсальный механизм индексирования. Они обеспечивают очень хорошее время доступа при масштабировании, возвращая данные из индекса по 100000 строк практически за то же время, что и из индекса по 1000 строк.

При проектировании надо уделить внимание тому, когда создавать индекс и по каким столбцам. Наличие индекса не всегда ускоряет доступ: во многих случаях оказывается, что при использовании индексов сервером Oracle производительность падает. Все зависит от того, к насколько большой части таблицы необходимо обратиться по индексу и как физически расположены данные. Если ответ на запрос можно полностью найти в индексе, доступ по индексу к большей части таблицы имеет смысл, поскольку позволяет избежать дополнительных операций чтения вразброс при обращении к таблице. Если же индекс используется для доступа к таблице, необходимо убедиться, что обрабатывается лишь небольшая часть таблицы.

Проектированием и созданием индексов надо заниматься по ходу разработки приложения, а не после ее завершения (как часто происходит). При правильном планировании и учете способов доступа к данным, как правило, понятно, какие индексы необходимы.

Индексы на основе битовых карт

Индексы на основе битовых карт появились в версии 7.3 сервера Oracle. Сейчас они доступны в редакциях Oracle 8i Enterprise и Personal Edition, но не в Standard Edition. Индексы на основе битовых карт создавались для хранилищ данных или сред с произвольными запросами, где полный список возможных запросов к данным при реализации приложения не полностью известен. Они не походят для систем OOT или систем, где данные часто изменяются несколькими одновременно работающими сеансами.

Индексы на основе битовых карт — это структуры, в которых хранятся указатели на множество строк, соответствующих одному значению ключа индекса, тогда как в структуре B*-дерева количество ключей индекса обычно примерно соответствует количеству строк. В индексе на основе битовых карт записей очень мало, и каждая из них указывает на множество строк. В индексе на основе B*-дерева обычно имеется однозначное соответствие — запись индекса ссылается на одну строку.

Предположим, создается индекс на основе битовых карт по столбцу **JOB** в таблице **EMP:**

```
scott@TKYTE816> create BITMAP index job_idx on emp(job);
Index created.
```

Сервер Oracle будет хранить в индексе примерно следующее:

Значение/Строка	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ANALYST	0	0	0	0	0	0	0	1	0	1	0	0	1	**0**
CLERK	1	0	0	0	0	0	0	0	0	0	1	1	0	1
MANAGER	0	0	0	1	0	1	1	0	0	0	0	0	0	0
PRESIDENT	0	0	0	0	0	0	0	0	1	0	0	0	0	0
SALESMAN	0	1	1	0	1	0	0	0	0	0	0	0	0	0

Это показывает, что в строках 8, 10 и 13 находится значение ANALYST, тогда как в строках 4, 6 и 7 — значение MANAGER. Также понятно, что пустых строк нет (индексы на основе битовых карт содержат записи для пустых значений — отсутствие такой записи в индексе означает, что пустых строк нет). Если необходимо посчитать, в скольких строках хранится значение MANAGER, индекс на основе битовых карт позволит сделать это очень быстро. Если необходимо найти все строки, в которых в столбце JOB хранится значение CLERK или MANAGER, достаточно просто скомбинировать соответствующие битовые карты из индекса:

Значение/Строка	1	2	3	4	5	6	7	8	9	10	//	12	13	14
CLERK	1	0	0	0	0	0	0	0	0	0	1	1	0	1
MANAGER	0	0	0	1	0	1	1	0	0	0	0	0	0	0
CLERK или MANAGER	1	0	0	1	0	1	1	0	0	0	1	1	0	1

Это позволяет быстро понять, что критериям поиска удовлетворяют строки 1, 4, 6, 7, 11, 12 и 14. Битовая карта, которую сервер Oracle хранит для каждого значения ключа, устроена так, что каждая позиция представляет идентификатор строки базовой таблицы на случай, если понадобится выбрать для дальнейшей обработки соответствующую строку. На запросы вида:

```
select count(*) from emp where job = 'CLERK' or job = 'MANAGER'
```

можно ответить непосредственно по индексу на основе битовых карт. Для ответа на запрос вида:

```
select * from emp where job = 'CLERK' or job = 'MANAGER'
```

придется обратиться к таблице. Сервер Oracle применит функцию, преобразующую установленный бит i в битовой карте в идентификатор строки, по которому можно обратиться к таблице.

Когда имеет смысл использовать индекс на основе битовых карт?

Индексы на основе битовых карт больше подходят для данных с небольшим количеством уникальных значений. Это данные, для которых при делении количества уникальных значений в строках на общее количество строк получается небольшое число (близкое к нулю). Например, столбец **GENDER** (пол) может иметь значения **M**, **F** и **NULL.** При наличии таблицы с 20000 записей о сотрудниках, получаем **3/20000 = 0,00015.** Этот столбец отлично подходит для создания индекса на основе битовых карт. Он, определенно, не подходит для создания индекса на основе B*-дерева, поскольку для каждого значения ключа будет извлекаться существенная часть строк таблицы. В общем случае, как было показано выше, индексы на основе B*-дерева должны быть избирательными. Индексы на основе битовых карт не должны быть избирательными, наоборот, они должны быть очень "неуникальными".

Индексы на основе битовых карт особенно хорошо подходят для сред с множеством произвольных запросов, особенно, если запросы эти ссылаются произвольным образом на много столбцов или выбирают агрегированные значения типа **COUNT.** Предположим, имеется таблица с тремя столбцами: **GENDER, LOCATION и AGE_GROUP. В** этой таблице столбец **GENDER** имеет значение **M** или **F**, столбец **LOCATION** может иметь значения от 1 до 50, а в столбце **AGE_GROUP** находится код, представляющий возрастные группы не старше 18 лет, 19-25 лет, 26-30 лет, 31-40 лет, 41 год и старше. Необходимо обеспечить выполнение множества произвольных запросов вида:

```
Select count(*)
  from T
 where gender = 'M'
   and location in (1, 10, 30)
   and age_group = '41 год и старше';

select *
  from t
 where ((gender = 'M' and location = 20)
        or (gender = 'F' and location = 22))
   and age_group = 'не старше 18 лет';

select count(*) from t where location in (11,20,30);

select count(*) from t where age_group = '41 год и старше' and gender = 'F';
```

Оказывается, обычная схема индексирования на основе B*-дерева тут не поможет. Если хочется использовать для получения ответа индексы, придется использовать от трех до шести возможных комбинаций индексов на основе B*-дерева для обеспечения доступа к данным. Поскольку в запросах может появиться любой из трех столбцов и любое подмножество трех столбцов, придется создать большие составные индексы на основе B*-дерева по следующим столбцам.

- **GENDER, LOCATION, AGE_GROUP.** Для запросов, использующих все три столбца, столбцы **GENDER** и **LOCATION** или только столбец **GENDER.**

- **LOCATION, AGE_GROUP.** Для запросов, использующих столбцы **LOCATION** и **AGE_GROUP** или только столбец **LOCATION.**
- **AGE_GROUP, GENDER.** Для запросов, использующих столбцы **AGE_GROUP** и **GENDER** или только столбец **AGE_GROUP**.

Чтобы сократить объем просматриваемых данных, имеет смысл проиндексировать и другие перестановки — это позволит сократить размер просматриваемых индексных структур. Не говоря уже о том, что создание индекса на основе B*-дерева по данным с таким небольшим количеством различных значений — вообще не лучшая идея.

Вот тут и пригодится индекс на основе битовых карт. С помощью трех небольших индексов на основе битовых карт, по одному для каждого отдельного столбца, можно эффективно находить строки, удовлетворяющие всем представленным выше условиям. Сервер Oracle будет просто объединять битовые карты трех индексов с помощью функций **AND, OR** и **XOR,** чтобы найти результирующее множество для условия, ссылающегося на любое подмножество этих трех столбцов. Он возьмет затем полученную в результате битовую карту, при необходимости преобразует биты со значением 1 в соответствующие идентификаторы строк и получит соответствующие данные (если бы требовалось выдать количество строк, удовлетворяющих условию, серверу достаточно было бы посчитать биты со значением 1).

Бывают случаи, когда индексы на основе битовых карт **не** подходят. Они хорошо работают в среде с интенсивным считыванием данных, но абсолютно не подходят для интенсивных изменений. Причина в том, что одна запись индекса на основе битовых карт ссылается на **множество** строк. Если сеанс изменяет проиндексированные данные, все строки, на которые ссылается соответствующая запись индекса, по сути оказываются заблокированными. Сервер Oracle не может заблокировать отдельный бит в битовой карте записи индекса; он блокирует всю битовую карту. Все остальные транзакции, которым необходимо изменить ту же битовую карту, будут заблокированы. Это существенно снижает степень параллелизма: в процессе каждого изменения потенциально блокируются сотни строк, что предотвращает одновременное изменение их столбцов, входящих в ключ индекса. Заблокированы будут не **все** строки, как можно было бы подумать, но многие. Битовые карты хранятся по разделам. Используя представленный выше пример с таблицей **EMP,** можно выяснить, что значение ключа индекса ANALYST появляется в индексе много раз, каждый раз указывая на сотни строк. При изменении строки, затрагивающем столбец **JOB,** необходимо получить исключительный доступ к двум записям индекса — для **старого** и **нового** значений ключа. Сотни строк, на которые указывают эти две записи, будут недоступны для изменения другим сеансам, пока исходное изменение не будет зафиксировано.

"Если сомневаетесь — пробуйте". Добавьте индекс (или несколько индексов) на основе битовых карт для таблицы и посмотрите, что это даст. Это не займет много времени, поскольку индексы на основе битовых карт создаются намного быстрее, чем индексы на основе B*-дерева. Экспериментирование — лучший способ узнать, подходят ли эти индексы для вашей среды. Меня часто спрашивают: "Как определить, что количество различных значений достаточно мало?". Простого ответа на этот вопрос нет. Иног-

да это 3 значения для 100000 строк. Иногда — 10000 значений для 1000000 строк. "Мало" не означает обязательно не более десятка различных значений. Именно эксперименты помогут понять, подходят ли индексы на основе битовых карт для приложения. В общем случае, при наличии используемой в основном для чтения большой таблицы, к которой выполняется много запросов, набор индексов на основе битовых карт может пригодиться.

Индексы по функциям

Индексы по функциям были добавлены в версии сервера Oracle 8.1.5. Они поддерживаются сейчас в редакциях Oracle8i Enterprise и Personal Edition, но не в Standard Edition.

Индексы по функциям позволяют индексировать вычисляемые столбцы и эффективно использовать их в запросах. По сути они позволяют реализовать не зависящий от регистра символов поиск или сортировку, искать результаты вычисления сложных выражений и эффективно расширять возможности языка SQL, добавляя собственные функции, а затем эффективно осуществляя по ним поиск.

Индексы по функциям имеет смысл использовать по многим причинам. Вот только основные из них.

- Индексы по функциям легко добавить, и они дают немедленный результат.
- Индексы по функциям можно использовать для ускорения работы существующих приложений, не изменяя логику их работы и запросы.

Важные детали реализации

Чтобы использовать индексы по функциям, необходима предварительная настройка. В отличие от описанных ранее индексов на основе B*-дерева и битовых карт, перед созданием и использованием индексов по функциям необходимо выполнить определенные действия. Чтобы обеспечить возможность их создания, надо задать ряд параметров в файле **init.ora** или на уровне сеанса. Кроме того, необходимы соответствующие привилегии. Для использования индексов по функциям необходимо сделать следующее.

- Чтобы создать индексы по функциям для таблиц в собственной схеме, необходима системная привилегия **QUERY REWRITE.**
- Чтобы создать индексы по функциям для таблиц в других схемах, необходима системная привилегия **GLOBAL QUERY REWRITE.**
- Использовать оптимизатор, основанный на стоимости. Индексы по функциям доступны только стоимостному оптимизатору, и никогда не будут использоваться оптимизатором на основе правил.
- Использовать функцию **SUBSTR,** чтобы ограничить размер значений типа **VARCHAR2** или **RAW,** возвращаемых пользовательскими функциями. Можно скрыть использование **SUBSTR** с помощью представления (рекомендуется так и делать). Соответствующие примеры представлены ниже.

- Чтобы оптимизатор использовал индексы по функциям, необходимо установить следующие параметры на уровне сеанса или системы:

```
QUERY_REWRITE_ENABLED=TRUE
QUERY_REWRITE_INTEGRITY=TRUSTED
```

  Необходимо установить эти параметры либо на уровне сеанса с помощью оператора ALTER SESSION, либо на уровне системы в файле параметров инициализации **init.ora.** Смысл установки параметра **QUERY_REWRITE_ENABLED** — разрешить оптимизатору переписывать запрос так, чтобы можно было использовать индекс по функции. Смысл установки параметра **QUERY_REWRITE_INTEGRITY** — сообщить оптимизатору, что можно "доверять" указанному программистом признаку предопределенности_результатов выполнения кода (deterministic). (Примеры кода с предопределенным результатом выполнения и смысл этой предопределенности рассматривается ниже.) Если результаты выполнения кода не предопределены (другими словами, возвращает разные результаты при одних и тех же входных данных), полученные по индексу строки могут оказаться некорректными. Предопределенность должен обеспечить разработчик.

После того как все пункты представленного выше списка выполнены, использовать индексы по функции просто — достаточно выполнить оператор **CREATE INDEX.** Оптимизатор найдет и использует эти индексы автоматически.

Пример использования индекса по функции

Рассмотрим следующий пример. Необходимо выполнить поиск, независимо от регистра символов, по столбцу **ENAME** таблицы **EMP.** До появления индексов по функции эта задача решалась по-другому. Приходилось добавлять дополнительный столбец в таблицу **EMP,** например **UPPER_ENAME.** Значения в этом столбце поддерживались с помощью триггера на события **INSERT** и **UPDATE,** который просто устанавливал **:NEW.UPPER_NAME := UPPER(:NEW.ENAME).** По этому дополнительному столбцу и создавался индекс. Теперь, при наличии индексов по функциям, этот дополнительный столбец не нужен.

Начнем с создания копии демонстрационной таблицы EMP в пользовательской схеме **SCOTT** и добавления в нее множества данных.

```
tkyte@TKYTE816> create table emp
  2  as
  3  select * from scott.emp;
Table created.

tkyte@TKYTE816>  set timing on
tkyte@TKYTE816> insert into emp
  2  select rownum EMPNO,
  3         substr(object_name,1,10) ENAME,
  4         substr(object_type,1,9) JOB,
  5              -rownum MGR,
  6              created hiredate,
```

```
  7                rownum SAL,
  8                rownum COMM,
  9                (mod(rownum,4)+1)*10  DEPTNO
 10    from all_objects
 11   where rownum < 10000
 12   /

9999 rows created.

Elapsed: 00:00:01.02
tkyte@TKYTE816> set timing off

tkyte@TKYTE816> commit;
Commit complete.
```

Теперь изменим данные в столбце фамилии сотрудника так, чтобы они хранились в смешанном регистре. Затем создадим индекс по функции UPPER от столбца ENAME, создавая по сути индекс, не зависящий от регистра символов в строке:

```
tkyte@TKYTE816> update emp set ename = initcap(ename);
10013 rows updated.

tkyte@TKYTE816> create index emp_upper_idx on emp(upper(ename));
Index created.
```

Наконец, проанализируем таблицу, поскольку, как уже было сказано, для использования индексов по функции надо применять оптимизатор, основанный на стоимости:

```
tkyte@TKYTE816> analyze table emp compute statistics
  2   for table
  3   for all indexed columns
  4   for all indexes;

Table analyzed.
```

Теперь имеется индекс по функции UPPER от столбца. Любое приложение, использующее не зависящие от регистра запросы и выполняющееся с соответствующими установками на уровне системы или сеанса, например:

```
tkyte@TKYTE816> alter session set QUERY_REWRITE_ENABLED=TRUE;
Session altered.

tkyte@TKYTE816> alter session set QUERY_REWRITE_INTEGRITY=TRUSTED;
Session altered.

tkyte@TKYTE816> set autotrace on explain
tkyte@TKYTE816> select ename, empno, sal from emp where upper(ename)='KING';

ENAME       EMPNO      SAL

King         7839     5000


Execution  Plan

    0       SELECT  STATEMENT  Optimizer=CHOOSE  (Cost=2  Card=9  Bytes=297)
    1    0    TABLE ACCESS  (BY INDEX ROWID) OF  'EMP'  (Cost=2  Card=9 Bytes=297)
    2    1      INDEX  (RANGE SCAN)  OF 'EMP_UPPER_IDX' (NON-UNIQUE) (Cost=l Card=9)
```

будет использовать этот индекс, что значительно повысит производительность. До появления подобных индексов нужно было просматривать каждую строку в таблице **EMP,** переводить значение столбца в верхний регистр и сравнивать с литералом. Теперь, при наличии индекса по **UPPER(ENAME),** сервер ищет литерал **KING** по индексу, просматривая несколько блоков данных, а затем обращается к таблице по идентификатору строки для получения соответствующих данных. Это делается очень быстро.

Рост производительности особенно заметен при индексировании по заданным пользователем функциям от столбцов. Начиная с версии Oracle 7.1, появилась возможность использовать в операторах SQL функции, задаваемые пользователем:

```
SQL> select my_function(ename)
  2   from emp
  3  where some_other function(empno)  > 10
  4  /
```

Это замечательно, потому что появилась возможность эффективного расширения языка SQL специфическими функциями приложения. К сожалению, однако, производительность при выполнении подобных запросов иногда крайне низка. Предположим, в таблице **EMP** - 1000 строк; тогда функция **SOME_OTHER_FUNCTION по** ходу выполнения запроса будет вызываться 1000 раз — по одному разу для каждой строки. Если функция выполняется, например, сотую долю секунды, то этот сравнительно простой запрос будет выполняться не менее 10 секунд.

Вот реальный пример. Я реализовал аналог функции **SOUNDEX** в языке PL/SQL Мы используем глобальную переменную пакета в качестве счетчика для процедуры — при выполнении запросов, использующих функцию **MY_SOUNDEX**, можно будет определить, сколько раз она вызывалась:

```
tkyte@TKYTE816> create  or  replace package  stats
  2   as
  3              cnt number default  0;
  4   end;
  5   /
Package created.

tkyte@TKYTE816> create or replace
  2   function my_soundex(p_string in varchar2)  return varchar2
  3   deterministic
  4   as
  5        l_return_string varchar2(6)  default substr(p_string,  1,  1) ;
  6        l_char        varchar2(1) ;
  7        l_last_digit      number default  0;
  8
  9        type vcArray is  table of varchar2(10)  index by binary_integer;
 10        l_code_table      vcArray;
 11
 12  begin
 13        stats.cnt  := stats.cnt+1;
 14
 15       l_code_table(1)  :=  'BPFV';
 16       1 code table(2)  :=  'CSKGJQXZ';
```

```
17        l_code_table(3)  :=  'DT';
18         l_code_table(4)  :=  'L;
19        l_code_table (5)  :=  'MN;
20         l_code_table(6)  :=  'R';
21
22
23         for  i  in  1  . .  length(p string)
24         loop
25              exit  when  (length (l_retum_string)=6);
26             l_char  := upper(substr( p_string,  i,  1 ));
27
28             for  j  in  1  .. l_code_table. count
29              loop
30              if  (instr(l_eode_table(j),  l_char)  > 0 AND j O l_last_digit)
31              then
32                   l_return_string  :=  l_return_string  ||  to_char(j,'fm9');
33           l_last_digit           := j;
34            end if;
35              end loop;
36        end  loop;
37
38       return  rpad(l_return_string,  6,  '0');
39   end;
40   /
```

Function created.

Обратите внимание, что в этой функции использовано новое ключевое слово, **DETERMINISTIC.** Оно означает, что данная функция при одних и тех же входных данных всегда даст одинаковый результат. Это необходимо указать при создании индекса по функции, заданной пользователем. Необходимо сообщить серверу Oracle, что результат выполнения функции предопределен **(DETERMINISTIC)** и она будет всегда давать одинаковые результаты при одинаковых входных данных. Это ключевое слово тесно связано с установкой **QUERY_REWRITE_INTEGRITY=TRUSTED** на уровне системы или сеанса. Мы сообщаем серверу Oracle, что можно быть уверенным в одинаковости результатов функции при одних и тех же входных данных, независимо от последовательности вызовов. В противном случае при доступе к данным по индексу и путем полного просмотра таблицы могли бы получаться разные результаты. Предопределенность означает, например, что нельзя создавать индекс по функции **DBMS_RANDOM.RANDOM** генератору случайных чисел — при тех же входных данных она дает случайные результаты. Результат встроенной функции SQL, **UPPER,** использованной в первом примере, предопределен, поэтому по функции **UPPER** от столбца индекс создать можно.

Теперь давайте разберемся, какую производительность будет иметь функция **MY_SOUNDEX** при отсутствии индекса. Используем созданную ранее таблицу **EMP** содержащую примерно 10000 строк:

```
tkyte@TKYTE816>  KEM  reset  our  counter
tkyte@TKYTE816>  exec  stats.cnt  :=  0
```

```
PL/SQL procedure successfully completed.

tkyte@TKYTE816> set timing on
tkyte@TKYTE816> set autotrace on explain
tkyte@TKYTE816> select ename, hiredate
  2     from emp
  3    where my_soundex(ename)  = my_soundex('Kings')
  4   /

ENAME   HIREDATE
------- ---------
King    17-NOV-81

Elapsed: 00:00:04.57

Execution  Plan

    0      SELECT STATEMENT Optimizer=CHOOSE  (Cost=12 Card=101 Bytes=16
    1    0   TABLE ACCESS  (FULL)  OF  'EMP'  (Cost=12 Card=101 Bytes=1616)

tkyte@TKYTE816> set autotrace off
tkyte@TKYTE816> set timing off

tkyte@TKYTE816> set serveroutput on
tkyte@TKYTE816> exec dbms_output.put_line(stats.cnt);
20026

PL/SQL procedure successfully completed.
```

Итак, запрос выполняется (путем полного просмотра таблицы) более четырех секунд. Функция **MY_SOUNDEX** была вызвана более 20000 раз (как показывает счетчик), дважды для каждой строки. Давайте посмотрим, как добавление индекса по функции позволит ускорить работу.

Сначала создадим индекс следующим образом:

```
tkyte@TKYTE816> create index emp_soundex_idx on
  2   emp(substr(my_soundex(ename),1,6))
  3   /
Index created.
```

Обратите внимание, что в этом операторе создания индекса используется функция SUBSTR. Дело в том, что индексируется функция, возвращающая строку. Если бы индексировалась функция, возвращающая число или дату, эта функция SUBSTR не понадобилась бы. Применять функцию SUBSTR к заданным пользователем функциям, возвращающим строки, необходимо потому, что они возвращают данные типа **VARCHAR2(4000)**. Это слишком большое значение для индексирования — запись индекса должна помещаться в треть блока. Если попытаться обойтись без SUBSTR, будет получено следующее сообщение (в базе данных с размером блока 8 Кбайт):

```
tkyte@TKYTE816> create index emp_soundex_idx on emp(my_soundex(ename));
create index emp_soundex_idx on emp(my_soundex(ename))

ERROR at line 1:
ORA-01450: maximum key length (3218) exceeded
```

В базах данных с другим размером блока может быть выдано другое значение вместо 3218, но пока используются блоки размером менее 16 Кбайт, создать индекс по данным типа столбцу VARCHAR2(4000) не удастся.

Итак, чтобы проиндексировать заданную пользователем функцию, возвращающую строку, необходимо ограничить тип возвращаемого значения в операторе CREATE INDEX. В рассмотренном выше примере, поскольку функция MY_SOUNDEX возвращает не более 6 символов, мы выбираем подстроку из первых шести символов.

Теперь все готово для оценки производительности при наличии индекса. Проследим последствия добавления индекса при выполнении операторов INSERT, а также ускорение выполнения операторов SELECT. Пока индекса не было, запросы выполнялись более четырех секунд, а вставка 10000 строк потребовала около одной секунды. Протестируем снова:

```
tkyte@TKYTE816> REM reset counter
tkyte@TKYTE816> exec stats.cnt := 0
PL/SQL procedure successfully completed.

tkyte@TKYTE816> set timing on
tkyte@TKYTE816> truncate table emp;
Table truncated.

tkyte@TKYTE816> insert into emp
  2   select -rownum EMPNO,
  3           initcap( substr(object_name,1,10)) ENAME,
  4          substr(object_type,1.9) JOB,
  5           -rownum MGR,
  6           created hiredate,
  7           rownum SAL,
  8           rownum COMM,
  9           (mod(rownum,4)+1)*10 DEPTNO
 10    from all_objects
 11   where rownum < 10000
 12   union all
 13   select empno, initcap(ename), job, mgr, hiredate,
 14          sal, comm, deptno
 15     from scott.emp
 16   /
10013 rows created.

Elapsed: 00:00:05.07
tkyte@TKYTE816> set timing off

tkyte@TKYTE816> exec dbms_output.put_line(stats.cnt);
10013

PL/SQL procedure successfully completed.
```

Итак, на этот раз выполнение операторов **INSERT** потребовало около 5 секунд. Причина — дополнительные расходы ресурсов на поддержку нового индекса по функции **MY_SOUNDEX**, — как на обычные действия по поддержке индекса (добавление любого индекса замедляет вставки), так и на вызов хранимой функции 10013 раз, как показывает значение переменной stats.cnt.

Теперь, для проверки запроса проанализируем таблицу и убедимся, что заданы необходимые установки сеанса:

```
tkyte@TKYTE816> analyze table emp compute statistics
  2   for table
  3   for all indexed columns
  4   for all indexes;
Table analyzed.

tkyte@TKYTE816> alter session set QUERY_REWRITE_ENABLED=TRUE;
Session altered.

tkyte@TKYTE816> alter session set QUERY_REWRITE_INTEGRITY=TRUSTED;
Session altered.
```

а затем выполним запрос:

```
tkyte@TKYTE816> REM reset our counter
tkyte@TKYTE816> exec stats.cnt := 0
PL/SQL procedure successfully completed.

tkyte@TKYTE816> set timing on

tkyte@TKYTE816> select ename, hiredate
  2     from emp
  3    where substr(my_soundex(ename),1,6) = my_soundex('Kings')
  4   /

ENAME   HIREDATE
------- ---------------
King    17-NOV-81

Elapsed: 00:00:00.10

tkyte@TKYTE816> set timing off
tkyte@TKYTE816> set serveroutput on
tkyte@TKYTE816> exec dbms_output.put_line(stats.cnt);

PL/SQL procedure successfully completed.
```

Если сравнить оба примера (без индекса и с индексом), окажется:

Действие	*Без индекса*	*При наличииндекса*	*Различие*	*Времявыполнения*
Insert	1.02	5.07	4.05	примерно в 5 раз медленнее
Select	**4.57**	0.10	**4.47**	примерно в 46 раз быстрее

Отметим следующие существенные моменты.

- Для вставки 10000 записей понадобилось примерно в пять раз больше времени. Индексирование заданной пользователем функции обязательно снизит производительность выполнения вставок и некоторых изменений. Надо понимать, что любой индекс снижает производительность: я выполнил тот же тест без функции MY_SOUNDEX, проиндексировав столбец ENAME. При этом операторы INSERT

выполнялись примерно 2 секунды (дополнительный расход ресурсов связан не только с использованием PL/SQL-функции). Поскольку большинство приложений вставляет и изменяет по одной записи, а для вставки строки необходимо всего 5/10000 секунды, замедление в типичном приложении никто скорее всего и не заметит. Поскольку вставляется строка только один раз, выполнять функцию придется только один раз, а не тысячи раз при выполнении запросов к данным.

- Хотя вставка и выполняется в пять раз медленнее, запрос выполняется примерно в 47 раз быстрее. Функция MY_SOUNDEX вычислялась всего два раза вместо 20000. Производительность запроса при наличии индекса и без него несравнима. Кроме того, с ростом размера таблицы, запрос с полным просмотром будет выполняться все дольше. Запрос по индексу всегда будет выполняться примерно за одно и то же время, независимо от размера таблицы.
- В запросе приходится использовать функцию SUBSTR. Это не так удобно, как написать **WHERE MY_SOUNDEX(ename)=MY_SOUNDEX('King')**, но проблему эту легко обойти, как будет показано ниже.

Итак, вставки замедлились, но запрос выполняется удивительно быстро. Небольшое замедление вставок и изменений с лихвой компенсируется. Кроме того, если столбцы, используемые функцией MY_SOUNDEX, не изменяются, то изменения вообще не замедляются (функция MY_SOUNDEX вызывается только при изменении столбца ENAME).

Теперь давайте рассмотрим, как добиться, чтобы в запросе не надо было использовать функцию SUBSTR. Требование использовать SUBSTR может провоцировать ошибки: пользователи должны помнить о необходимости выбирать первые 6 символов с помощью SUBSTR. Если они укажут другой размер, индекс не будет использоваться. Хотелось бы также контролировать на сервере количество индексируемых байтов. Это позволило бы при необходимости в дальнейшем изменить функцию MY_SOUNDEX так, чтобы она возвращала 7 байт вместо 6. Это можно очень просто сделать, скрыв вызов SUBSTR с помощью представления:

```
tkyte@TKYTE816> create or replace view emp_v
  2  as
  3  select ename, substr(my_soundex(ename),1,6) ename_soundex, hiredate
  4     from emp
  5  /
View created.
```

Теперь можно выполнять запросы к представлению:

```
tkyte@TKYTE816> exec stats.cnt := 0;
PL/SQL procedure successfully completed.

tkyte@TKYTE816> set timing on

tkyte@TKYTE816> select ename, hiredate
  2     from emp_v
  3   where ename_soundex = my_soundex('Kings')
  4  /
```

```
ENAME      HIREDATE
---------- -----------------
King       17-NOV-81

Elapsed:  00:00:00.10

tkyte@TKYTE816> set timing off

tkyte@TKYTE816> exec dbms_output.put_line(stats.cnt)
2

PL/SQL procedure successfully completed.
```

Используется тот же план выполнения запроса, что и при обращении к базовой таблице. Мы просто скрыли вызов **SUBSTR(F(X), 1, 6)** в представлении. Оптимизатор все равно распознает, что этот виртуальный столбец на самом деле проиндексирован и делает "то, что нужно". Мы получили то же повышение производительности и тот же план выполнения запроса. Использование этого представления — ничем не хуже использования базовой таблицы, а даже лучше, поскольку позволяет скрыть сложности и в дальнейшем легко изменить размер строки, возвращаемой с помощью SUBSTR.

Подводный камень

При использовании индексов по функции я столкнулся с одной проблемой: не удается создать такой индекс по встроенной функции **TO_DATE**. Например:

```
ops$tkyte@ORA8I.WORLD> create index t2 on t(to_date(y,'yyyy'));
create index t2 on t(to_date(y,'yyyy'))
                                    *
ERROR at line 1:
ORA-01743: only pure functions can be indexed
```

Это известная ошибка, которая будет исправлена в следующих версиях Oracle (после 8.1.7). До этого придется создавать собственную интерфейсную функцию для встроенной функции TO_DATE и индексировать ее:

```
ops$tkyte@ORA8I.WORLD> create or replace
  2   function my_to_date(p_str in varchar2,
  3                       p_fmt in varchar2) return date
  4   DETERMINISTIC
  5   is
  6   begin
  7            return to_date( p_str, p_fmt );
  8   end;
  9   /
Function created.

ops$tkyte@ORA8I.WORLD> create index t2 on t(my_to_data(y,'yyyy'));
Index created.
```

Итак, индексы по функции просты в использовании и реализации, и их применение дает немедленный результат. Их можно использовать для ускорения работы существующих приложений, без изменения их алгоритмов или запросов. Можно получить уско-

рение работы на несколько порядков. Эти индексы можно использовать для предварительного вычисления сложных значений без использования триггера. Кроме того, оптимизатор может более точно оценивать избирательность, если результаты вычисления выражений хранятся в индексе по функции.

С другой стороны, в таблицу с индексом по функции, заданной пользователем и требующей обращения к SQL-машине, нельзя загружать данные в непосредственном режиме. Это означает, что нельзя выполнять непосредственную загрузку в таблицу, проиндексированную по **MY_SOUNDEX(X),** но можно, — если проиндексировано выражение **UPPER(x).**

Индексы по функции снижают производительность выполнения вставок и изменений данных. Существенно это или нет — в каждом конкретном случае решать разработчику. Если данные накапливаются, но запрашиваются редко, индексы по функции могут и не подойти. С другой стороны, помните, что обычно строки вставляются по одной, а запрашиваются тысячи раз. Снижение производительности при вставке (которое отдельный пользователь может и не заметить) может тысячекратно быть оправдано за счет ускорения выполнения запросов.

В общем случае преимущества индексов по функции существенно перевешивают недостатки.

Прикладные индексы

Прикладные индексы в базах данных Oracle обеспечивают расширяемое индексирование. Они позволяют создавать собственные индексные структуры, работающие аналогично стандартным индексам сервера Oracle. При выполнении оператора **CREATE INDEX,** использующего прикладной тип индекса, сервер Oracle будет выполнять соответствующий код создания индекса. Если проанализировать индекс для получения его статистической информации, сервер Oracle будет выполнять соответствующий код для генерации этих данных в том формате, который предполагался создателями индекса. Когда сервер Oracle анализирует запрос и вырабатывает план, который может использовать прикладной индекс, он "спросит" у прикладного кода: "Сколько будет стоить выполнение этой функции?", поскольку ему приходится оценивать разные планы. Если коротко, прикладные индексы позволяют создавать новые, еще не существующие в базе данных, типы индексов. Например, если создается приложение, анализирующее хранящиеся в базе данных изображения и выдающее информацию об этих изображениях — скажем, используемые цвета — можно создать специальный индекс по изображениям. При добавлении изображений в базу данных будет вызываться код для извлечения информации о цветах, которая будет сохраняться отдельно (там, где сочтет нужным разработчик). При выполнении запросов, требующих вернуть "изображения в синих тонах", сервер Oracle при необходимости потребует от прикладного индекса вернуть ответ.

Лучший пример — собственный текстовый индекс компонента **interMedia.** Этот индекс обеспечивает поиск по ключевым словам в больших текстах. Компонент **interMedia** предлагает собственный тип индекса:

```
ops$tkyte@ORA8I.WORLD> create  index  myindex  on  mytable(docs)
  2   indextype  is  ctxsys.context
  3  /
Index  created.
```

и отдельные специальные операторы в языке SQL:

```
select * from mytable where contains(docs,  'some  words')  > 0;
```

Этот индекс позволяет даже выполнять команды вида:

```
ops$tkyte@ORA8I.WORLD> analyze  index myindex compute  statistics;
Index  analyzed.
```

Он будет взаимодействовать с оптимизатором в ходе выполнения оператора, при определении относительной стоимости использования текстового индекса по сравнению с другим индексом или полным просмотром таблицы. Интересно, что такого рода индекс может разработать кто угодно. Реализация текстового индекса **interMedia** не использует внутренние особенности ядра. Все было сделано на основе открытых и описанных интерфейсов прикладных программ для создания такого рода структур. Ядро сервера Oracle "не знает", как хранится текстовый индекс **interMedia** (для хранения каждого создаваемого индекса используется несколько физических таблиц). Сервер Oracle "не знает" о том, что происходит при вставке новой строки. Компонент **interMedia** — это фактически приложение, построенное на основе базы данных, но полностью в нее интегрированное. Для пользователей оно не отличается от других средств ядра сервера Oracle, но на самом деле в ядро не входит.

Лично я не вижу особой нужды создавать новые экзотические типы индексных структур. Эту возможность используют в основном сторонние производители, предлагающие революционные методы индексирования. Например, компания Virage, Inc. использовала этот же функциональный интерфейс для реализации специального индекса в базах данных Oracle. Этот индекс позволяет индексировать загружаемые в базу данных изображения. Затем можно искать картинки, похожие на другие картинки по текстуре, цветам, освещению и т.п. Можно было бы создать индекс, позволяющий осуществлять поиск по отпечаткам пальцев, хранящимся в базе данных как большой двоичный объект, из внешнего приложения, считывающего отпечатки пальцев. Он мог бы хранить информацию о ключевых точках отпечатков в таблицах базы данных, кластерах или, возможно, в обычных внешних файлах — в зависимости от того, что лучше подойдет. После этого можно будет с помощью операторов SQL вводить отпечатки и сравнивать с отпечатками, хранящимися в базе данных, так же просто, как и числа: **SELECT * FROM T WHERE X BETWEEN 1 AND 2.**

Мне кажется, самое интересное в прикладных индексах — возможность добавлять новые технологии индексирования для приложений. Большинство пользователей никогда не будут использовать соответствующий функциональный интерфейс для создания нового типа индексов, но конечными результатами будут пользоваться многие. Практически во всех приложениях, над которыми мне приходилось работать, надо хранить и обрабатывать **текст,** данные в формате **XML** или **изображения.** Функциональные воз-

можности компонента **interMedia,** реализованные с помощью прикладных индексов, позволяют это сделать. Со временем набор доступных типов индексов растет. Например, в базу данных Oracle 8.1.7 добавлены *индексы Rtree* (индексы Rtree используются для индексирования пространственных данных).

Часто задаваемые вопросы об индексах

Как уже было сказано во введении, мне приходится отвечать на множество вопросов о СУБД Oracle. Я — именно тот Том, который ведет рубрику "AskTom" в журнале Oracle Magazine и поддерживает сайт **http://asktom.oracle.com,** где пользователи могут получить ответы на вопросы о базе данных и средствах разработки Oracle. Практика показывает, что наибольшее количество вопросов касается индексов. В этом разделе я дам ответы на некоторые из наиболее часто и постоянно задаваемых вопросов об индексах. Некоторые ответы могут показаться очевидными, другие — могут вас удивить. Надо сказать, что с индексами связано множество мифов и непонимания.

Работают ли индексы с представлениями?

Часто задают похожий вопрос: "Как проиндексировать представление?". Суть в том, что представление — это сохраненный запрос. Сервер Oracle будет подставлять в текст запроса к представлению определение самого представления. Представления обеспечивают удобство для конечных пользователей, оптимизатор же работает с запросом к базовым таблицам. Любые индексы, которые могли бы использоваться в запросе, непосредственно обращающемся к базовым таблицам, будут учтены при использовании представления. Чтобы проиндексировать представление, надо просто проиндексировать базовые таблицы.

Индексы и пустые значения

Индексы на основе B*-дерева, кроме индекса кластера, не содержат записей для полностью пустых значений, а индексы на основе битовых карт и индекс кластера — имеют. Этот побочный эффект может использоваться с выгодой, если понимать, что он означает.

Чтобы понять, как сказывается то, что значения **Null не хранятся,** рассмотрим следующий пример:

```
ops$tkyte@ORA8I.WORLD> create table t (x int, y int);
Table created.

ops$tkyte@ORA8I.WORLD> create unique index t_idx on t(x,y);
Index created.

ops$tkyte@ORA8I.WORLD> insert into t values (1, 1);
1 row created.

ops$tkyte@ORA8I.WORLD> insert into t values (1, NULL);
1 row created.
```

```
ops$tkyte@ORA8I.WORLD> insert into t values (NULL, 1);
1 row created.

ops$tkyte@ORA8I.WORLD> insert into t values (NULL, NULL);
1 row created.

ops$tkyte@ORA8I.WORLD> analyze index t_idx validate structure;
Index analyzed.

ops$tkyte@ORA8I.WORLD> select name, lf_rows from index_stats;

NAME                  LF ROWS
--------------- ------------
T_IDX                        3
```

В таблице — четыре строки, а в индексе — только три. Первые три строки, в которых хотя бы один из ключей индекса — не Null, входят в индекс. Последняя строка со значениями (NULL, NULL) в индекс не входит. Один из случаев, вызывающих непонимание, когда индекс — уникальный, как в примере выше. Рассмотрим результаты выполнения трех следующих операторов INSERT:

```
ops$tkyte@ORA8I.WORLD> insert into t values (NULL, NULL);
1 row created.

ops$tkyte@ORA8I.WORLD> insert into t values (NOLL, 1);
insert into t values ( NULL, 1 )
*
ERROR at line 1:
ORA-00001: unique constraint (OPS$TKYTE.T_IDX) violated


ops$tkyte@ORA8I.WORLD> insert into t values (1, NULL);
insert into t values ( 1, NULL )
*
ERROR at line 1:
ORA-00001: unique constraint (OPS$TKYTE.T_IDX) violated
```

Новая строка (NULL, NULL) не считается совпадающей со старой строкой со значениями (NULL, NULL):

```
ops$tkyte@ORA8I.WORLD> select x, y, count(*)
  2   from t
  3   group by x,y
  4   having count(*) > 1;

     X       Y   COUNT(*)
------  ------ ----------
```

Это кажется невероятным: уникальный ключ оказывается не уникальным, если все столбцы имеют пустые значения. Факт в том, что в СУБД Oracle (NULL, NULL) <> (NULL, NULL). Эти два ключа не совпадают при сравнении, но совпадают при группировке (при использовании конструкции GROUP BY). Учтите следующее: каждое требование *уникальности* должно включать хотя бы один непустой столбец, чтобы обеспечивать действительную уникальность.

Еще один часто задаваемый вопрос: "Почему запрос не использует индекс?" связан с индексами и пустыми значениями. При этом речь идет о запросе вида:

```
select * from T where x is null;
```

Этот запрос не может использовать созданный ранее индекс — строка (NULL, NULL) просто не входит в индекс, поэтому при использовании индекса был бы получен неправильный ответ. Запрос сможет использовать индекс, только если хотя бы для **одного** из столбцов задано требование **NOT NULL.** Например, можно показать, что сервер Oracle будет использовать индекс по столбцу при поиске по условию X IS **NULL,** если X — начальный столбец индекса, и хотя бы один из остальных столбцов, входящих в индекс, имеет требование **NOT NULL:**

```
ops$tkyte@ORA8I.WORLD> create table t  (x int, y int NOT NULL);
Table created.

ops$tkyte@ORA8I.WORLD> create unique index t_idx on t(x,y);
Index created.

ops$tkyte@ORA8I.WORLD> insert into t values  (1,  1) ;
1 row created.

ops$tkyte@ORA8I.WORLD> insert into t values  (NULL,  1) ;
1 row created.

ops$tkyte@ORA8I.WORLD> analyze table t compute statistics;
Table analyzed.

ops$tkyte@ORA8I.WORLD> set autotrace on
ops$tkyte@ORA8I.WORLD> select * from t where x is null;

      X        Y
 ------ --------
               1


Execution Plan
----------------------------------------------------------------------------
    0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=8)
    1    0    INDEX (RANGE SCAN) OF 'T_IDX' (UNIQUE) (Cost=1 Card=1 Bytes=8)
```

Я уже говорил, что можно использовать тот факт, что полностью пустые записи в индексе на основе B*-дерева не хранятся. Вот как это делать. Предположим у нас есть таблица со столбцом, имеющим всего два значения. Значения используются крайне неравномерно: допустим, 90 процентов строк содержат одно значение, а 10 процентов — другое. Можно эффективно проиндексировать этот столбец, чтобы получить доступ к строкам, составляющим меньшинство. Это пригодится, когда необходимо проиндексировать меньшую часть строк, но оставить возможность полного просмотра большинства строк, сэкономив при этом место. Решение состоит в том, чтобы использовать значение **Null** в большинстве строк и любое другое значение — в остальных строках.

Допустим, таблица используется в качестве своего рода "очереди". Пользователи вставляют в нее строки, которые должны обрабатываться другим процессом. Подавля-

ющее большинство строк в этой таблице находится в состоянии "обработано", и лишь немногие еще не обработаны. Можно создать таблицу следующего вида:

```
create table t ( ... другие столбцы .. ., timestamp DATE default SYSDATE);
create index t_idx on t(timestamp);
```

Теперь, при вставке новой строки, она будет помечена текущим временем. Процесс будет запрашивать данные с помощью такого запроса, использующего очень давнюю дату для получения всех текущих записей:

```
select * from T where timestamp > to_date('01010001','ddmmyyyyy') order by
timestamp;
```

И по мере обработки этих записей он будет изменять столбец **timestamp,** устанавливая в нем значение NULL, удаляя ее тем самым из индекса. Поэтому индекс по этой таблице остается небольшим по размеру, независимо от количества строк в таблице. Если есть вероятность, что некоторые записи долго не будут обработаны, т.е. в индексе могут быть "долго живущие" записи, можно потребовать физически освободить пространство и сжать индекс. Это можно сделать с помощью оператора **ALTER INDEX ... COALESCE. B** противном случае, индекс со временем будет излишне разрастаться (становиться менее плотным). Если строки всегда обрабатываются и удаляются из индекса, этот шаг не нужен.

Теперь, когда известно, как обрабатываются пустые значения в индексе на основе B*-дерева, можно использовать это с выгодой для приложения и учесть особенности ограничения уникальности по нескольким столбцам, все из которых могут быть пустыми (будьте готовы к тому, что в этом случае может быть несколько строк с пустыми значениями).

Индексы по внешним ключам

Часто задают вопрос, надо ли индексировать внешние ключи. Мы уже касались этой темы в главе 3, при обсуждении взаимных блокировок. Там я подчеркивал, что не проиндексированные внешние ключи являются, как правило, наиболее частой причиной возникновения взаимных блокировок, поскольку изменение в главной таблице или удаление записи из главной таблицы приводит в этом случае к блокированию всей подчиненной таблицы (никакие изменения в таблице внешнего ключа будут невозможны, пока транзакция не завершится). При этом блокируется намного больше строк, чем нужно, и снижается параллелизм. Я часто видел, как это происходит, когда используются средства, автоматически генерирующие SQL-операторы для изменения таблицы. Генерируется оператор **UPDATE,** изменяющий все столбцы таблицы, независимо от того, изменено значение в столбце или нет. При этом изменяется первичный ключ (хотя на самом деле его значение не изменяется никогда). Например, Oracle Forms будет делать это по умолчанию, если не потребовать явно передавать в базу данных только измененные столбцы. Помимо блокирования таблицы, не проиндексированный внешний ключ плох еще и в следующих случаях:

- При наличии конструкции ON DELETE CASCADE. Например, таблица EMP является подчиненной для таблицы DEPT. Оператор DELETE FROM DEPT WHERE DEPTNO = 10 должен вызвать каскадное удаление в таблице EMP. Если столбец DEPTNO в таблице EMP не проиндексирован, для этого придется выполнить полный просмотр таблицы EMP. Этот полный просмотр нежелателен; кроме того, при удалении большого количества строк из главной таблицы подчиненная будет каждый раз полностью просматриваться.

- При выполнении запроса от главной таблицы к подчиненной. Рассмотрим пример с таблицами EMP/DEPT еще раз. Очень часто таблица EMP запрашивается с условием по столбцу **DEPTNO**. Если приходится часто выполнять запрос:

```
select *
  from dept, emp
  where emp.deptno = dept.deptno
  and dept.dname = :X;
```

  для генерации отчета или других целей, окажется, что отсутствие индекса существенно замедляет выполнение запросов. Этот же аргумент я приводил, обосновывая необходимость индексировать столбец **NESTED_COLUMN_ID** вложенной таблицы. Скрытый столбец **NESTED_COLUMN_ID** вложенной таблицы — это просто внешний ключ.

Итак, когда не нужно индексировать внешний ключ? Если выполнены следующие условия:

- данные из главной таблицы не удаляются;
- значение первичного/уникального ключа главной таблицы не изменяется ни намеренно, ни случайно (используемым инструментальным средством);
- не выполняется соединение от главной таблицы к подчиненной, т.е. столбцы внешнего ключа не обеспечивают важный способ доступа к подчиненной таблице (как в случае таблиц DEPT и EMP).

Если все три условия выполняются, индекс можно не создавать: он только замедлит выполнение операторов ЯМД. Если же какие-то из перечисленных действий выполняются, помните о последствиях.

Если предполагается, что подчиненная таблица блокируется из-за того, что не проиндексирован внешний ключ и необходимо в этом убедиться (или предотвратить), можно выполнить команду:

```
ALTER TABLE <имя подчиненной таблицы> DISABLE TABLE LOCK;
```

Теперь оператор **UPDATE** или **DELETE**, примененный к главной таблице и вызывающий блокирование подчиненной таблицы, приводит к выдаче сообщения:

```
ERROR at line 1:
ORA-00069: cannot acquire lock — table locks disabled for <имя подчиненной
   таблицы>
```

Это пригодится при поиске фрагмента кода, делающего то, что не должен (например, изменять или удалять первичный ключ), поскольку пользователи моментально сообщат вам об этом.

Почему мой индекс не используется?

Для этого может быть много причин; мы рассмотрим наиболее типичные.

Случай 1

Используется индекс на основе B*-дерева, и в условии не используются начальные столбцы ключа индекса. В этом случае есть таблица T и индекс по T(x,y). Выполняется запрос **SELECT * FROM T WHERE Y = 5.** Оптимизатор скорее всего не будет использовать этот индекс, поскольку в условии не упоминается столбец X, — пришлось бы просматривать все записи индекса. Скорее всего будет выбран полный просмотр таблицы T. Это не исключает использования индекса в принципе. Если бы выполнялся запрос SELECT X,Y FROM T WHERE Y = 5, оптимизатор учел бы, что для получения значений X и Y обращаться к таблице не придется (эти столбцы входят в индекс), и выбрал бы быстрый просмотр самого индекса, поскольку листовой уровень индекса обычно намного меньше базовой таблицы. Учтите также, что этот способ доступа поддерживает только оптимизатор, основанный на стоимости.

Случай 2

Используется запрос **SELECT COUNT(*) FROM T** (или аналогичный), и есть индекс на основе B*-дерева по таблице T. Однако оптимизатор выбирает полный просмотр таблицы вместо подсчета записей индекса (намного меньших). В этом случае индекс, очевидно, создан по столбцам, которые могут содержать пустые значения. Поскольку полностью состоящие из пустых значений записи в индекс не попадают, количество строк в индексе не будет совпадать с количеством строк в таблице. В данном случае оптимизатор поступает правильно: если бы для подсчета строк использовался индекс, результат был бы неправильным.

Случай 3

Запрос обращается к проиндексированному столбцу:

```
select * from t where f(indexed_column) = value
```

и оказывается, что индекс по столбцу **INDEX_COLUMN** не используется. Это происходит потому, что используется функция от столбца. Индексировались значения столбца **INDEX_COLUMN**, а не значения **F(INDEXED_COLUMN).** Индекс здесь не поможет. Если надо, можно проиндексировать функцию от столбца.

Случай 4

Проиндексирован столбец символьного типа. Этот столбец содержит только числовые данные. Выполняется следующий запрос:

```
select * from t where indexed_column = 5
```

Обратите внимание на число 5 в запросе: это числовой литерал 5 (а не символьная строка). Индекс по столбцу **INDEXED_COLUMN** не используется. Дело в том, что представленный выше запрос эквивалентен следующему:

```
select * from t where to_number(indexed_column) = 5
```

К столбцу неявно применяется функция, и, как было показано в случае 3, это не позволяет применить индекс. В этом очень легко убедиться на следующем примере:

```
ops$tkyte@ORA8I.WORLD> create table t (x char(1) primary key) ;
Table created.

ops$tkyte@ORA8I.WORLD> insert into t values ('5');
1 row created.

ops$tkyte@ORA8I.WORLD> set autotrace on explain

ops$tkyte@ORA8I.WORLD> select * from t where x = 5;

X
-
5

Execution Plan
----------------------------------------------------------------------
    0        SELECT STATEMENT Optimizer=CHOOSE
    1     0    TABLE ACCESS (FULL) OF "I"


ops$tkyte@ORA8I.WORLD> select * from t where x = '5',

X
-
5

Execution Plan
----------------------------------------------------------------------
    0        SELECT STATEMENT Optimizer=CHOOSE
    1     0    INDEX (UNIQUE SCAN) OF 'SYS_C0038216' (UNIQUE)
```

Как бы то ни было, неявных преобразований надо избегать всегда. Всегда сравнивайте яблоки с яблоками, а апельсины — с апельсинами. Часто такая ситуация наблюдается с датами. Вы пытаетесь выполнить запрос:

```
— найти все записи за сегодня
select * from t where trunc(date_col) = trunc(sysdate);
```

Оказывается, что индекс по столбцу **DATE_COL** не используется. Можно либо проиндексировать функцию **TRUNC(DATE_COL)**, либо, что проще, переписать запрос с помощью оператора сравнения **BETWEEN**. Следующий пример демонстрирует использование условия **BETWEEN** для дат. Достаточно понять, что условие:

```
TRUNC (DATE_COL) = TRUNC (SYSDATE)
```

эквивалентно условию:

```
DATE_COL BETWEEN TRUNC(SYSDATE) AND TRUNC (SYSDATE) ПЛЮС ОДНИ СУТКИ МИНУС
   ОДНА СЕКУНДА
```

После этого понятно, как использовать конструкцию **BETWEEN.**

```
select *
  from t
  where date_col between trunc(sysdate) and trunc(sysdate)+1-1/(1*24*60*60)
```

*Примечаще: выражение **1/(1*24*60*60)** означает часть суток, равную одной секунде. Вычитание 1 отбросит один день, 1/24 — один час, а 1/(24*60) — одну минуту.*

При этом все функции переносятся в правую часть оператора сравнения, что позволяет использовать индекс по столбцу **DATE_COL** (по сути условие эквивалентно исходному, **WHERE TRUNC(DATE_COL) = TRUNC(SYSDATE)). По возможности, надо избегать применения функций к столбцам базы данных в условиях.** Это позволит не только использовать индексы, но и сократит объем вычислений, которые необходимо выполнять серверу. В предыдущем случае, когда используется условие:

```
between trunc(sysdate) and trunc(sydate)+1/(1*24*60*60)
```

значения вычисляются в запросе только один раз, а затем можно просто искать соответствующие значения ключа по индексу. При использовании условия **TRUNC(DATE_COL) = TRUNC(SYSDATE)** выражение **TRUNC(DATE_COL)** придется вычислять **для каждой строки** во всей таблице (индекс ведь не используется).

Случай 5

При использовании индекса работа только замедляется. Я видел это часто: разработчики предполагают, что индекс всегда ускоряет выполнение запроса. Поэтому они создают небольшую таблицу, анализируют ее и обнаруживают, что оптимизатор не использует индекс. В данном случае оптимизатор делает правильный выбор. Сервер Oracle (при использовании оптимизатора, основанного на стоимости) будет использовать индекс, только если в этом есть смысл. Рассмотрим следующий пример:

```
ops$tkyte@ORA8I.WORLD> create table t
  2   (x, y null, primary key (x))
  3  as
  4   select rownum x, username
  5     from all users
  6    where rownum <= 100
  7  /
Table created.

ops$tkyte@ORA8I.WORLD> analyze table t compute statistics;
Table analyzed.

ops$tkyte@ORA8I.WORLD> analyze table t compute statistics for all indexes;
Table analyzed.
```

```
ops$tkyte@ORA8I.WORLD> set autotrace on explain
ops$tkyte@ORA8I.WORLD> select count(y)  from t where x < 50;

  COUNT(Y)

        49


Execution  Plan

    0      SELECT STATEMENT Optimizer=CHOOSE  (Cost=1 Card=1 Bytes=18)
    1    0   SORT  (AGGREGATE)
    2    1     TABLE ACCESS  (FULL)  OF  'T'  (Cost=1 Card=50 Bytes=900)
```

Оптимизатор, основанный на стоимости, обращается к таблице и определяет, что придется выбирать 50 процентов ее строк. Делать это по индексу медленно; придется читать блок индекса, а затем обрабатывать все строки, на которые он ссылается, причем для каждой строки придется читать блок базы данных. Намного эффективнее просто прочитать все строки в блоке и найти те 50 процентов, которые надо обрабатывать. Теперь, если немного изменить пример:

```
ops$tkyte@ORA8I.WORLD> set autotrace off
ops$tkyte@ORA8I.WORLD> insert into t
  2  select rownum+100, username
  3     from all_users
  4  /
41231 rows created.

ops$tkyte@ORA8I.WORLD> analyze table t compute statistics;
Table analyzed.

ops$tkyte@ORA8I.WORLD> analyze table t compute statistics for all indexes;
Table analyzed.

ops$tkyte@ORA8I.WORLD> set autotrace on explain
ops$tkyte@ORA8I.WORLD> select count(y)  from t where x < 50;

  COUNT(Y)

        49


Execution  Plan

    0      SELECT STATEMENT Optimizer=CHOOSE  (Cost=3 Card=1 Bytes=21)
    1    0   SORT  (AGGREGATE)
    2    1     TABLE ACCESS  (BY INDEX ROWID)  OF  'I'  (Cost=3 Card=50
    3    2       INDEX  (RANGE SCAN)  OF  'SYS_C0038226'  (UNIQUE)  (Cost=2
```

Оптимизатор "понимает", что теперь по условию будет выбираться около 0,1 процента строк, поэтому, определенно, есть смысл использовать индекс.

Этот пример показывает, во-первых, что индексы должны использоваться **не всегда.** Прежде чем делать выводы, убедитесь, что доступ по индексу действительно будет выполняться быстрее. А во-вторых, насколько важна актуальная статистическая информа-

ция. Если после загрузки большого количества данных не проанализировать таблицы, оптимизатор будет принимать ошибочные решения, что и приводит к случаю 6.

Случай 6

Таблицы некоторое время не анализировались; раньше они были сравнительно небольшими, а сейчас заметно выросли. Теперь имеет смысл использовать индекс, который ранее ничего не давал. Если проанализировать таблицу, индекс будет использоваться. Возвращаясь к предыдущему примеру, но, выполняя запрос до и после вставки строк, можно это явно продемонстрировать:

```
ops$tkyte@ORA8I.WORLD> insert  into  t
   2   select rownum+100, username
   3      from all_users
   4   /
41231 rows created.

ops$tkyte@ORA8I.WORLD> set autotrace on explain
ops$tkyte@ORA8I.WORLD> select count(y)  from t where x < 50;

  COUNT(Y)

         49


Execution  Plan

    0       SELECT STATEMENT Optimizer=CHOOSE  (Cost=1 Card=1 Bytes=18)
    1    0    SORT  (AGGREGATE)
    2    1      TABLE ACCESS  (FULL)  OF  'T'  (Cost=1 Card=50 Bytes=900)

ops$tkyte@ORA8I.WORLD> set autotrace off

ops$tkyte@ORA8I.WORLD> analyze table t compute statistics;
Table analyzed.

ops$tkyte@ORA8I.WORLD> analyze table t compute statistics for all indexes;
Table analyzed.

ops$tkyte@ORA8I.WORLD> set autotrace on explain
ops$tkyte@ORA8I.WORLD> select count(y)  from t where x < 50;

  COUNT(Y)

         49


Execution  Plan

    0       SELECT STATEMENT Optimizer=CHOOSE  (Cost=3 Card=1 Bytes=21)
    1    0   SORT  (AGGREGATE)
    2    1     TABLE ACCESS  (BY INDEX ROWID)  OF  'T'  (Cost=3 Card=50 Bytes=1050)
    3    2       INDEX  (RANGE SCAN)  OF  'SYS_C0038227'  (UNIQUE)  (Cost=2 Card=50)
```

При отсутствии актуальной статистической информации оптимизатор, основанный на стоимости, **не может** принимать правильные решения.

По моему опыту, эти шесть случаев демонстрируют **основные** причины, по которым не используются индексы. Все в конечном итоге сводится к тому, что "их нельзя использовать, потому что это даст неверные результаты" или "их нет смысла использовать, потому что это снизит производительность".

Использовались ли индексы?

На этот вопрос ответить сложно. Сервер Oracle не отслеживает обращения к индексам, так что нельзя просто посчитать записи в *проверочной таблице* (audit trail table) или предложить аналогичное по простоте решение. Для решения этой проблемы простого способа нет. В версии Oracle 8i, однако, можно использовать два подхода.

В главе 11 я буду описывать, как *хранимые шаблоны запросов* (stored query outlines) — средство сохранения подсказок для выбора плана выполнения запроса сервером Oracle в таблице базы данных — могут использоваться для определения того, какие индексы используются. Можно включить хранение (но не использование) планов. При этом можно записать все планы для всех выполненных запросов, не изменяя приложения. Затем можно выполнить запрос к таблицам шаблонов и определить, какие методы доступа по индексу использовались, и даже попытаться выяснить, для каких именно запросов использовались индексы.

Еще один метод — поместить каждый индекс в отдельное табличное пространство или в отдельный файл в табличном пространстве. Сервер Oracle отслеживает ввод/вывод в каждый файл (доступ к этой **информации можно** получить через представление динамической производительности **V$FILESTAT).** Если для табличного пространства с индексом количество чтений **примерно соответствует** количеству записей, понятно, что этот индекс не используется для доступа к данным. Он читался сервером, **только** когда изменялся (сервер Oracle изменяет индекс при выполнении операторов **INSERT, UPDATE** и **DELETE).** Если табличное пространство практически **не читается,** понятно, что индекс не используется и таблица изменяется не часто. Если же чтений выполняется **больше,** чем записей, — это свидетельствует о том, что индекс используется.

Недостаток этих подходов в том, что, даже если удастся разобраться, какие индексы использовались, а какие — нет, все равно непонятно, **правильный** ли это набор индексов для имеющихся данных. Эти подходы не подскажут, что при простом добавлении столбца X к индексу Y можно избежать обращения к таблице по идентификатору строки и тем самым существенно повысить эффективность запроса. Один из общих способов оптимизации состоит в добавлении столбцов в конец индекса так, чтобы для ответов на запросы использовался только индекс и к таблице обращаться вообще не пришлось. Так нельзя понять, что есть избыточные индексы. Например, при наличии индексов по T(X), T(X,Y) и T(X,Y,Z), первые два, вероятно, можно удалить, не снизив при этом производительности запросов и ускорив выполнение изменений, хотя бывают случаи, когда это неверно. Идея в том, что хорошо спроектированная и документированная система должна периодически проверять, используются ли имеющиеся индек-

сы, поскольку последствия наличия индексов надо продумывать заранее для всей системы в целом, а не только для отдельных запросов. В быстро изменяющейся системе с большим количеством разработчиков, постоянно подключающихся и уходящих из проекта, это обычно не делается.

Миф: пространство в индексе никогда повторно не используется

Этот миф я хочу развеять раз и навсегда: пространство в индексе **используется** повторно.

Миф этот возникает следующим образом: имеется таблица T со столбцом X. В некоторый момент времени в таблицу добавляется строка со значением X = 5. Затем вы ее удаляете. Миф состоит в том, что пространство, выделенное для записи X = 5, не будет использоваться повторно до тех пор, пока в индекс опять не будет вставлена запись со значением X = 5. Миф утверждает, что, как только слот индекса использован, он остается занятым навсегда, и может использоваться только под такое же значение. Этот миф дополняется мифом о том, что освободившееся в индексе пространство никогда индексной структуре не возвращается, и блок индекса никогда не используется повторно. Это тоже неправда.

Опровергнуть первую часть мифа просто. Достаточно создать следующую таблицу:

```
tkyte@ORA8I.WORLD> create table t (x int, constraint t_pk primary key(x));
Table created.

tkyte@ORA8I.WORLD» insert into t values (1);
1 row created.

tkyte@ORA8I.WORLD> insert into t values (2);
1 row created.

tkyte@ORA8I.WORLD> insert into t values (9999999999);
1 row created.

tkyte@ORA8I.WORLD> exec show_space('T_PK', user, 'INDEX');
Free Blocks.....0
Total Blocks.....64
Unused Blocks.....62

PL/SQL procedure successfully completed.
```

Итак, в соответствии с мифом, если выполнить оператор **delete from T where x = 2,** это пространство не будет использоваться повторно до тех пор, пока не будет повторно вставлено значение 2. Сейчас индекс использует два блока: один — для карты экстентов, второй — для данных индекса. Если при удалении записи индекса никогда повторно не используются, а я буду вставлять и удалять строки, не используя прежние значения, то индекс должен чрезмерно разрастись. Давайте посмотрим:

```
ops$tkyte@ORA8I.WORLD>begin
  2           for i in 2 .. 999999
  3            loop
  4                     delete from t where x = i;
```

```
  5                       commit;
  6                      insert into t values  (i+1);
  7                       commit;
  8            end loop;
  9  end;
 10  /

PL/SQL procedure successfully completed.

ops$tkyte@ORA8I.WORLD> exec show_space('T_PK', user,  'INDEX');
Free Blocks. . . . . .0
Total Blocks. . . . . .64
Unused Blocks. . . . . .62

PL/SQL procedure successfully completed.
```

Итак, это показывает, что пространство в индексе повторно используется. Как и в большинстве мифов, однако, доля правды в мифе все-таки есть. Правда в том, что пространство, выделенное первоначальному значению 2 (в диапазоне от 1 до 9999999999), останется в этом блоке индекса навсегда. Индекс сам себя не "уплотняет". Это означает, что если загрузить в таблицу значения от 1 до 500000, а затем удалить, скажем, все строки с нечетными значениями, в индексе по этому столбцу будет 250000 "дырок". Только при повторной вставке данных, которые попадают в блок с дыркой, соответствующее пространство будет повторно использовано. Сервер Oracle не будет пытаться сжать или уменьшить индекс. Это можно сделать с помощью операторов ALTER INDEX REBUILD или COALESCE. С другой стороны, если загрузить в таблицу значения от 1 до 500000, а затем удалить все строки со значениями, меньшими 250000, окажется, что освободившиеся блоки индекса снова возвращены в список свободных мест индекса. Это пространство полностью может быть использовано повторно. Если помните, второй миф утверждал как раз обратное. Утверждалось, что выделенное индексу пространство никогда не "возвращается". В соответствии с этим мифом, после того как блок индекса использован, он оказывается в соответствующем месте структуры индекса навсегда и будет использован повторно только при вставке данных, которые должны были бы попасть в это место структуры индекса. Можно показать, что и это неправда. Сначала построим таблицу с примерно 500000 строк:

```
ops$tkyte@ORA8I.WORLD> create table t
  2   ( x int ) ;
Table created.

ops$tkyte@ORA8I.WORLD> insert /*+ APPEND */ into t select rownum from
all_objects;
30402 rows created.

ops$tkyte@ORA8I.WORLD> commit;
Commit complete.

ops$tkyte@ORA8I.WORLD> insert /*+ APPEND */ into t
  2  select rownum+cnt from t,  (select count(*)  cnt from t) ;
30402 rows created.

ops$tkyte@ORA8I.WORLD> commit;
Commit complete.
```

```
ops$tkyte@ORA8I.WORLD> insert /*+ APPEND */ into t
  2  select rownum+cnt from t, (select count(*) cnt from t) ;
60804 rows created.

ops$tkyte@ORA8I.WORLD> commit;
Commit complete.

ops$tkyte@ORA8I.WORLD> insert /*+ APPEND */ into t
  2  select rownum+cnt from t, (select count(*) cnt from t) ;
121608 rows created.

ops$tkyte@ORA8I.WORLD> commit;
Commit complete.

ops$tkyte@ORA8I.WORLD> insert /*+ APPEND */ into t
  2  select rownum+cnt from t, (select count(*) cnt from t) ;
243216 rows created.

ops$tkyte@ORA8I.WORLD> commit;
Commit complete.

ops$tkyte@ORA8I.WORLD> alter table t add constraint t_pk primary key(x)
  2  /
Table altered.
```

Теперь определим, сколько пространства используется до и после массового удаления данных:

```
ops$tkyte@ORA8I.WORLD> exec show_space('T_PK', user, 'INDEX');
Free Blocks.....0
Total Blocks.....1024
Unused Blocks.....5

PL/SQL procedure successfully completed.

ops$tkyte@ORA8I.WORLD> delete from t where x < 250000;
249999 rows deleted.

ops$tkyte@ORA8I.WORLD> commit;
Commit complete.

ops$tkyte@ORA8I.WORLD> exec show_space('T_PK', user, 'INDEX');
Free Blocks.....520
Total Blocks.....1024
Unused Blocks.....5

PL/SQL procedure successfully completed.
```

Как видите, более половины блоков индекса теперь находятся в списке свободных. Это означает, что блоки — абсолютно пустые (блоки в списке свободных мест индекса должны быть полностью свободны, в отличие от блоков в списке свободных мест таблицы, организованной в виде кучи).

Итак, вам продемонстрировали следующее.

- Пространство в блоках индекса используется повторно, когда добавляется строка, которая может его повторно использовать.

- Пустой блок индекса удаляется из индексной структуры и может повторно использоваться в дальнейшем. Происхождение мифа, вероятно, связано с тем, что наличие свободного пространства в блоках индекса в индексной структуре не видно так, как в таблице. В таблице блок может оказаться в списке свободных, даже если в нем есть данные. В список свободных блоков индекса попадают только абсолютно пустые блоки; блоки, в которых находится только одна запись индекса (а остальное пространство свободно), выявить непросто.

Миф: столбцы с максимальным количеством разных значений должны указываться первыми

Кажется, это следует из соображений здравого смысла. Если предполагается создание индекса по столбцам C1, C2 таблицы со 100000 строк, при этом столбец C1 имеет 100000 уникальных значений, а столбец C2 — 25000, индекс создается по столбцам T(C1,C2). Это означает, что столбец C1 должен указываться первым, что соответствует "здравому смыслу". Фактически при сравнении векторов данных (пара значений C1, C2 задает вектор) порядок столбцов не имеет значения. Рассмотрим следующий пример. Создадим таблицу со всеми объектами базы данных, а затем — индекс по столбцам OWNER, OBJECT_TYPE и OBJECT_NAME (начиная со столбца с минимальным количеством значений) и индекс по столбцам OBJECT_NAME, OBJECT_TYPE и OWNER:

```
tkyte@TKYTE816> create table t
  2  nologging
  3  as
  4  select * from all_objects;
Table created.

tkyte@TKYTE816> create index t_idx_1 on t(owner,object_type,object_name)
  2  nologging pctfree 0;
Index created.

tkyte@TKYTE816> create index t_idx_2 on t(object_name,object_type,owner)
  2  nologging pctfree 0;
Index created.

tkyte@TKYTE816> select count(distinct owner), count(distinct object_type),
  2   count(distinct object_name) , count(*)
  3  from t;

 (DISTINCT OWNER) (DISTINCT OBJECT_TYPE) (DISTINCT OBJECT_NAME) COUNT(*)

               24                     23                  12265      21975
```

Теперь, чтобы показать, что по используемому пространству ни один из индексов не имеет преимущества, определим, сколько пространства они используют:

```
tkyte@TKYTE816> exec show_space('T_IDX_1', user, 'INDEX');
Free Blocks.....0
Total Blocks.....192
```

```
Total Bytes.....1572864
Unused Blocks.....51
Unused Bytes.....417792
Last Used Ext FileId.....6
Last Used Ext BlockId.....4745
Last Used Block.....13
PL/SQL procedure successfully completed.
tkyte@TKYTE816> exec show_space('T_IDX_2', user, 'INDEX');
Free Blocks.....0
Total Blocks.....192
Total Bytes.....1572864
Unused Blocks.....51
Unused Bytes.....417792
Last Used Ext FileId.....6
Last Used Ext BlockId.....4937
Last Used Block.....13

PL/SQL procedure successfully completed.
```

Они используют одинаковый объем пространства. Однако первый индекс куда лучше будет **сжиматься,** если использовать сжатие ключей. Есть аргумент в пользу перечисления столбцов, начиная с наименее избирательного. Теперь сравним производительность: посмотрим, является ли какой-то из индексов более эффективным в этом смысле. Чтобы протестировать это, я использовал блок PL/SQL с запросами, включающими подсказки (так, чтобы использовался тот или иной индекс) следующего вида:

```
tkyte@TKYTE816> alter session set sql_trace=true;
Session altered.
tkyte@TKYTE816> declare
  2            cnt int;
  3  begin
  4            for x in (select owner, object_type, object_name from t)
  5            loop
  6         select /*+ INDEX(t t_idx_1) */ count(*) into cnt
  7           from t
  8          where object_name = x.object_name
  9            and object_type = x.object_type
 10            and owner = x.owner;
 11
 12         select /*+ INDEX(t t_idx_2) */ count(*) into cnt
 13           from t
 14          where object_name = x.object_name
 15            and object_type = x.object_type
 16            and owner = x.owner;
 17            end loop;
 18  end;
 19  /

PL/SQL procedure successfully completed.
```

Эти запросы читают по индексу все строки таблицы. Отчет, сгенерированный утилитой **TKPROF,** показывает следующее:

```
SELECT /*+ INDEX(t t_idx_1)   */COUNT(*)
FROM
 T   WHERE OBJECT_NAME =  :bl   AND OBJECT_TYPE =  :b2  AND OWNER =  :b3


call      count      cpu    elapsed    disk      query  current      rows

Parse          1 0 . 0 0        0.00       0          0          0         0
Execute   21975     2.35        2.55       0          0          0         0
Fetch     21975     1.40        1.57       0      44088          0     21975

total     43951     3.75        4.12       0      44088          0     21975

Rows      Execution Plan

      0  SELECT STATEMENT    GOAL: CHOOSE
  21975    SORT  (AGGREGATE)
  21975     INDEX  (RANGE SCAN)  OF  'T_IDX_1'  (NON-UNIQUE)
********************************************************************************

SELECT /*+ INDEX(t t_idx_2)   */COONT(*)
FROM
 T   WHERE OBJECT_NAME =  :b1  AND OBJECT_TYPE =  :b2  AND OWNER =  :b3


call      count      cpu     elapsed    disk      query  current     rows

Parse          1 0 . 0 0        0.00       0          0          0         0
Execute   21975     2.10        2.44       0          0          0         0
Fetch     21975     1.65        1.60       0      44088          0     21975

total     43951     3.75        4.04       0      44088          0     21975

Rows      Execution Plan
--------  ---------------------------------------------------
      0  SELECT STATEMENT    GOAL: CHOOSE
  21975    SORT  (AGGREGATE)
  21975     INDEX  (RANGE SCAN)  OF  'T_IDX_2'  (NON-UNIQUE)
```

Итак, в обоих случаях обработано одинаковое количество строк, блоков, затрачено одинаковое процессорное время; совпадает и реальное время выполнения (выполните этот тест еще раз, и значения CPU и ELAPSED будут немного отличаться, но в среднем они одинаковы). Никаких существенных преимуществ перечисление столбцов в порядке уменьшения избирательности не дает.

Решение поместить столбец C1 перед столбцом C2 должно определяться тем, как используется индекс. Если имеется множество запросов вида:

```
select * from t where c1 = :x and c2 = :y;
select * from t where c2 = :y;
```

имеет смысл создать индекс по столбцам T(C2,C1) — один этот индекс можно использовать для выполнения обоих запросов. Кроме того, с помощью сжатия ключей (кото-

рое мы уже рассматривали в контексте таблиц, организованных по индексу, и позже рассмотрим еще), можно создать меньший по размеру индекс, если столбец C2 указан первым. Дело в том, что значения столбца C2 повторяются в индексе в среднем четыре раза. Если оба столбца, C1 и C2, имеют средний размер 10 байт, записи этого индекса займут не более 2000000 байт (100000 * 20). Прибегнув к сжатию ключей индекса по столбцам (C2, C1), можно сжать индекс до 1250000 (100000 * 12,5) байт, поскольку 3 из 4 повторяющихся значений столбца C2 можно убрать.

В Oracle 5 (да, в версии 5), была одна причина указывать наиболее избирательные столбцы в индексе первыми. Это было связано с тем, как в версии 5 реализовывалось сжатие индексов (не путайте со сжатием ключа индекса). Эта возможность была убрана в версии 6 и одновременно добавлено блокирование на уровне строк. После этого указание наиболее избирательных столбцов в индексе первыми уже не позволяет уменьшить индекс или сделать его более эффективным. С учетом возможности сжатия ключей, появился убедительный аргумент для того, чтобы поступать как раз наоборот, поскольку это позволяет уменьшить индекс. Однако все зависит от того, как используется индекс.

Резюме

В этой главе описаны различные типы индексов, поддерживаемые в Oracle. Мы начали с простого индекса на основе B*-дерева, затем рассмотрели подтипы этого индекса, такие как индексы с обращенным ключом, созданные для среды Oracle Parallel Server, и индексы по убыванию — для извлечения данных упорядоченными по возрастанию по одному столбцу, и по убыванию — по другому. Мы выяснили, когда стоит использовать индекс и почему в некоторых случаях индекс не помогает.

Затем мы рассмотрели индексы на основе битовых карт — прекрасный метод индексирования для данных с небольшим и средним количеством уникальных значений в среде хранилищ данных (где данные интенсивно считываются, но редко изменяются). Мы разобрались, когда имеет смысл использовать индекс на основе битовых карт и почему нет смысла их использовать в среде оперативной обработки транзакций, да и в любой среде, где несколько пользователей должны одновременно изменять один и тот же столбец.

Потом мы перешли к индексам по функциям, которые являются отдельным случаем индексов на основе B*-дерева и битовых карт. Индексы по функциям позволяют создавать индекс по выражению над столбцами, т.е. предварительно вычислять и запоминать результаты вычисления сложных выражений и заданных пользователем функций, чтобы в дальнейшем быстро выбирать их по индексу. Мы рассмотрели ряд важных особенностей реализации индексов по функциям, в частности необходимые для их использования установки на уровне системы или сеанса. После этого были рассмотрены примеры создания индексов как по встроенным функциям Oracle, так и по функциям, заданным пользователями. Наконец, мы рассмотрели проблему, возникающую при использовании индексов по встроенной функции Oracle TO_DATE и способ обойти ее.

Затем мы рассмотрели очень специфический тип индексов — прикладные индексы. Вместо того чтобы создавать подобный индекс с нуля (что долго и сложно), мы рассмотрели пример использования уже реализованного прикладного индекса **interMedia Text**. К этому очень важному типу индексов мы еще вернемся в специально посвященной ему главе 17.

Завершается глава ответами на часто задаваемые мне вопросы об индексах. От простейших: "работают ли индексы для представлений", до более сложных: "правда ли, что пространство в индексе не используется повторно". Ответы на эти вопросы давались с использованием примеров, демонстрирующих суть проблем.

8

Импорт и экспорт

Утилиты импорта (IMP) и экспорта (EXP) можно отнести к старейшим инструментальным средствам Oracle. Это утилиты командной строки, используемые для извлечения таблиц, схем или всей базы данных из одного экземпляра Oracle для дальнейшего импортирования в другой экземпляр или схему.

Традиционно утилиты импорта и экспорта принято относить к сфере интересов администратора базы данных. Я же считаю, что они полезнее как инструмент разработчика, а не средство администрирования. Поскольку изменился как размер, так и значение баз данных, изменились и средства управления ими. В прошлом было вполне разумно использовать утилиты импорта и экспорта для пересоздания базы данных (скажем, с целью изменения размера блока или переноса базы данных на другую платформу) или в качестве средства резервного копирования. Сегодня, когда маленькими считаются базы данных размером порядка гигабайт, простые средства, выполняющие обработку последовательно (как утилиты импорта и экспорта), просто недостаточно масштабируемы. Хотя эти средства и не бесполезны для администратора базы данных, их применимость сейчас значительно меньше. На смену им пришли другие средства и методы. Например, сейчас для выполнения инкрементного резервного копирования больших баз данных используется диспетчер восстановления (Recovery Manager — RMAN), а не утилита EXP. Но эти утилиты по-прежнему используются для решения других задач, например для выявления логического и физического повреждения данных, для переноса файлов данных из базы и т.д.

Утилиты EXP и IMP вам придется использовать рано или поздно. Если необходимо скопировать схему одного пользователя другому, проще всего это сделать с помощью утилит EXP и IMP. Если необходимо получить операторы ЯОД, образующие схему, —

используйте EXP и IMP. Если нужно разбить существующую таблицу на несколько физических фрагментов (что требует пересоздания таблицы), утилиты EXP и IMP вполне подойдут для этой цели. В этой главе мы рассмотрим, как:

- выделять подмножества данных;
- получить операторы ЯОД, использованные для создания объектов схемы базы данных;
- импортировать небольшие и средние объемы данных в другие структуры (отличные от тех, где они находились);
- клонировать схему в базе данных; это кажется несложным, но есть ряд тонкостей, о которых следует знать.

Утилиты EXP и IMP будут рассматриваться с точки зрения разработчика. Я предполагаю ответить на множество часто задаваемых вопросов о практическом использовании утилит EXP и IMP. Я сам использую эти средства и вынужден был решать различные проблемы, возникающие при их использовании.

Простой пример

Чтобы продемонстрировать значимость утилит IMP и EXP и простоту их использования, получим с их помощью оператор ЯОД, который позволит воссоздать таблицу в схеме SCOTT/TIGER. Множество пользователей ищут утилиты для решения такой задачи или пытаются создать собственные средства извлечения операторов ЯОД, — они не знают, что такая утилита поставляется вместе с сервером. Вот как просто это делается:

```
C:\ImpExp>exp userid=scott/tiger tables=emp

C:\ImpExp>imp userid=scott/tiger full=y indexfile=emp.sql

C:\ImpExp>type emp.sql

REM CREATE TABLE "SCOTT"."EMP" ("EMPNO" NUMBER(4, 0) KOT NULL ENABLE,
REM "ENAME" VARCHAR2(10) , "JOB" VARCHAR2 (9) , "MGR" NUMBER(4, 0),
REM "HIREDATE" DATE, "SAL" NUMBER (7, 2), "COMM" NUMBER (7, 2), "DEPTNO"
REM NUMBER(2, 0)) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 LOGGING
REM STORAGE (INITIAL 32768 NEXT 32768 MINEXTENTS 1 MAXEXTENTS 4096
REM PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
REM TABLESPACE "TOOLS" ;
REM ... 14 rows
CONNECT SCOTT;
CREATE UNIQUE INDEX "SCOTT"."EMP_PK" ON "EMP" ("EMPNO" ) PCTFREE 10
INITRANS 2 MAXTRANS 255 STORAGE(INITIAL 32768 NEXT 32768 MINEXTENTS 1
MAXEXTENTS 4096 PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL
DEFAULT) TABLESPACE "TOOLS" LOGGING ;
REM ALTER TABLE "SCOTT"."EMP" ADD CONSTRAINT "EMP_PK" PRIMARY KEY
REM ("EMPNO") USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255
REM STORAGE (INITIAL 32768 NEXT 32768 MINEXTENTS 1 MAXEXTENTS 4096
```

```
REM  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
REM  TABLESPACE "TOOLS" ENABLE ;
REM  ALTER TABLE "SCOTT"."EMP" ADD CONSTRAINT "EMP_FK_DEPT" FOREIGN KEY
REM   ("DEPTNO") REFERENCES "DEPT" ("DEPTNO") ENABLE NOVALIDATE;
REM  ALTER TABLE "SCOTT"."EMP" ADD CONSTRAINT "EMP_FK_EMP" FOREIGN KEY
REM   ("MGR") REFERENCES "EMP" ("EMPNO") ENABLE NOVALIDATE ;
REM  ALTER TABLE "SCOTT"."EMP" ENABLE CONSTRAINT "EMP_PK" ;
REM  ALTER TABLE "SCOTT"."EMP" ENABLE CONSTRAINT "EMP_FK_DEPT" ;
REM  ALTER TABLE "SCOTT"."EMP" ENABLE CONSTRAINT "EMP_FK_EMP" ;
```

Вот оно, со всей пресловутой многословностью (у вас результат может отличаться, поскольку здесь использованы "улучшения", сделанные мной в таблице **SCOTT.EMP** по ходу написания книги). Получился оператор ЯОД, позволяющий пересоздать таблицу **EMP** в текущем состоянии. Немного позже мы рассмотрим ряд особенностей использования утилит IMP и EXP для успешного извлечения операторов ЯОД из базы данных (а также сценарии, которые пригодятся, если возможностей IMP/EXP окажется недостаточно).

Полная документация по утилитам EXP и IMP включена в руководство "Oracle Server Utilities Guide". Считая бессмысленным повторять представленную там информацию, я рекомендую прочитать соответствующие разделы этого руководства в качестве дополнения к данной главе.

Утилита EXP создает двоичный файл специфического формата, который также называют файлом дампа (и часто сокращенно ссылаются на него как на **DMP**). Этот формат допускает перенос в разные операционные системы — его можно перенести с ОС Windows 2000 на Sun Solaris, с Sun на MVS, и он везде будет работать.

Это также означает, что нельзя использовать утилиту EXP для выгрузки данных из Oracle с целью дальнейшего импортирования в SQL Server — она для этого не подходит. Если это необходимо сделать, можно использовать команду **copy** утилиты SQL*Plus в сочетании либо со шлюзом Transparent Gateway to SQL Server, либо, что намного проще, с драйверами Net8 для ODBC, входящими в состав Oracle Developer/2000 (эти драйверы позволяют утилите SQL*Plus подключаться к источнику данных ODBC). При этом нет надобности редактировать файл **DMP** с помощью текстового (да и любого другого) редактора. Файл **DMP** имеет одно единственное назначение — он предназначен для чтения и обработки утилитой IMP.

Когда могут пригодиться утилиты IMP и EXP?

Утилиту EXP можно использовать для многих целей, некоторые из них упоминались. Ниже я перечислю ряд ситуаций, когда она оказалась особенно полезной.

Выявление повреждений

Я использую утилиту EXP в качестве средства для выявления повреждений базы данных, физических или логических. Если с помощью утилиты EXP выполнить *полное* эк-

спортирование базы данных, она проверит весь словарь данных и найдет в нем любую логическую несогласованность. Кроме того, она полностью просмотрит каждую таблицу в базе данных, читая все строки. Если имеется таблица со сбойным блоком, этот блок будет найден. Утилита не выявляет определенные типы логических повреждений, например индекс, указывающий на несуществующие строки, поскольку просто просматривает таблицы, но обнаружит наиболее существенные ошибки (индекс всегда можно пересоздать, а вот пересоздание таблицы иногда невозможно).

Поскольку утилита EXP полностью читает таблицу, она также находит блоки, требующие очистки. Это дополнительный шанс предотвратить спорадические ошибки **ORA-01555** (подробнее об этом см. в главе 5). После экспортирования всей базы данных и проверки результатов (поиска ошибок в журналах), я делаю полный импорт в режиме **SHOW.** Эта процедура имеет интересный побочный эффект: создает большой журнальный файл, содержащий все операторы ЯОД, текст всех процедур, триггеров, представлений и т.д. В экстренных случаях я неоднократно использовал этот журнал для восстановления пропавшего фрагмента кода. Кроме того, если таблица "случайно" удалена, можно быстро восстановить ее из файла **DMP,** не обращаясь к реальным резервным копиям. Если есть место для хранения файлов **DMP** (они очень хорошо упаковываются), я рекомендовал бы выполнять экспортирование базы данных в часы минимальной нагрузки. В разделе "Экспортирование больших объемов данных" будет представлен сценарий для UNIX-систем, экспортирующий данные непосредственно в сжатые файлы, что позволяет экономить много места.

Извлечение операторов ЯОД

Утилита EXP — замечательное средство извлечения операторов ЯОД, формирующих базу данных (как было показано в главе 6). С ее помощью очень просто получить подробный оператор **CREATE** для многих объектов. В разделе "Получение операторов ЯОД" мы подробно рассмотрим эту возможность.

Клонирование схем

Утилиты EXP и IMP можно использовать для клонирования схемы с целью тестирования. С помощью опций **FROMUSER** и **TOUSER** команды IMP данные с легкостью переносятся от одного пользователя к другому. Это также официально поддерживаемый метод "переименования" пользователя: необходимо экспортировать схему пользователя, импортировать эти данные в схему нового пользователя и, проверив успешность операции, удалить учетную запись старого пользователя.

Перенос табличных пространств

Утилиты EXP и IMP могут использоваться для "переноса" табличного пространства или набора табличных пространств. Эта возможность, доступная только в Oracle 8i, позволяет взять сформатированные файлы данных из одного экземпляра и "подключить" их к другому. Рассмотрим ситуацию, когда на общедоступном Web-сайте необходимо опубликовать большой каталог. Можно создать каталог с внутренней стороны брандма-

уэра, настроить и протестировать его. Затем для публикации каталога можно просто экспортировать соответствующее табличное пространство и все связанные с ним пространства (со всеми служебными структурами наподобие индексов) и скопировать полученные файлы данных на серверы вне брандмауэра. Не нужно больше "сбрасывать и загружать" данные для их публикации — переключение со старого каталога на новый выполняется очень быстро. Преимущества использования этого приема в среде хранилищ данных очевидны: вместо классического процесса ETL (Extract, Transform, Load — извлечь, преобразовать, загрузить) можно выполнить только L (Load — загрузить), присоединив файлы данных из оперативной базы к хранилищу данных и использовав для преобразования операторы SQL. Я также интенсивно использовал эту возможность в среде тестирования и разработки. Чтобы протестировать программное обеспечение, всегда приходится "сбрасывать" базу данных. С помощью переносимых табличных пространств это можно сделать быстро, без восстановления всей базы данных, что позволяет тестировать в одной и той же базе данных несколько разных проектов (они не будут сбрасывать данные друг друга). Таким образом, можно использовать один экземпляр базы данных, а не держать по экземпляру для каждого проекта.

Пересоздание экземпляров

Использование утилит EXP и IMP —эффективный способ пересоздания экземпляра небольшого размера. Если, например, необходимо изменить размер блока базы данных, утилиты EXP и **IMP** оказываются подходящим средством. Возможно, для экземпляров с большим объемом данных это неприемлемо из-за большой продолжительности процесса, но для систем с объемом данных в несколько гигабайт — это вариант. Я бы не стал применять такой подход для экземпляра с терабайтами данных (да и вообще для базы данных объемом более 15 гигабайт).

Копирование данных с одной платформы на другую

Утилиты EXP и IMP — прекрасное средство копирования данных с одной платформы на другую, даже путем пересылки их по электронной почте. Если создать файл **DMP** на одной платформе, его можно импортировать на любой другой — данные хранятся в виде, не зависящем от платформы, хотя файл **DMP** и двоичный.

Есть и другие варианты творческого использования этих утилит, но основные уже перечислены. Теперь я постараюсь ответить на часто задаваемые вопросы, возникающие при использовании утилит EXP и **IMP,** и опишу применение этих утилит.

Особенности использования утилит

В этом разделе мы рассмотрим многие из часто задаваемых по поводу утилит IMP и EXP вопросов типа "Как сделать...". Прежде всего я опишу наиболее существенные опции утилит и их назначение.

Опции

Параметры для утилит EXP и IMP задаются в виде пар имя-значение. Используется или такой вызов:

```
exp parameter_name = value
```

или:

```
exp parameter_name = (value1,value2,value3...)
```

Второй метод удобен для выполнения определенных операций, например, экспортирования на уровне таблиц, а также для экспортирования нескольких таблиц за раз. Опции утилит IMP и EXP можно задавать в файле параметров, чтобы не набирать их постоянно в командной строке.

Как EXP, так и IMP поддерживает опцию HELP = Y, которая выдает на экран краткую информацию об использовании. Она пригодится в том случае, когда требуется уточнить имя параметра. Если просто ввести в командной строке EXP или IMP и нажать клавишу *Enter,* утилиты будут запущены в "интерактивном" режиме и начнут поочередно запрашивать значения необходимых параметров.

Параметры утилиты EXP

Вот что выдаст EXP, если передать только параметр HELP = Y:

```
C:\exp>exp help=y

Export: Release 8.1.6.0.0 - Production on Mon Mar 19 14:11:23 2001

(c) Copyright 1999 Oracle Corporation.  All rights reserved.

You can let Export prompt you for parameters by entering the EXP
command followed by your username/password:

      Example: EXP SCOTT/TIGER

Or, you can control how Export runs by entering the EXP command followed
by various arguments. To specify parameters, you use keywords:

      Format:  EXP KEYWORD=value or KEYWORD=(value1,value2,...,valueN)
      Example: EXP SCOTT/TIGER GRANTS=Y TABLES=(EMP,DEPT,MGR)

                   or TABLES=(T1:P1,T1:P2), if T1 is partitioned table

USERID must be the first parameter on the command line.

Keyword    Description (Default)       Keyword      Description (Default)
USERID     username/password           FULL         export entire file (N)
BUFFER     size of data buffer         OWNER        list of owner usernames
FILE       output files (EXPDAT.DMP)   TABLES       list of table names
COMPRESS   import into one extent (Y)  RECORDLENGTH  length of 10 record
GRANTS     export grants (Y)           INCTYPE      incremental export type
INDEXES    export indexes (Y)          RECORD       track incr. export (Y)
```

```
ROWS         export data rows  (Y)       PARFILE      parameter filename
CONSTRAINTS  export constraints  (Y)     CONSISTENT   cross-table consistency
LOG          log file of screen output   STATISTICS   analyze objects
 (ESTIMATe)
DIRECT       direct path  (N)            TRIGGERS     export triggers  (Y)
FEEDBACK     display progress every x rows  (0)
FILESIZE     maximum size of each dump file
QUERY        select clause used to export a subset of a table

The following keywords only apply to transportable tablespaces
TRANSPORT_TABLESPACE export transportable tablespace metadata  (N)
TABLESPACES list of tablespaces to transport
Export terminated successfully without warnings.
```

Давайте подробно рассмотрим наиболее существенные параметры, а также те параметры, которые требуют комментариев. Очевидные параметры, вроде USERID, мы описывать не будем. Параметры, которые я считаю устаревшими, такие как INCTYPE, тоже не описываются:

Имя параметра	*Стандартное значение*	*Назначение/Примечания*
BUFFER	Зависит от ОС	Этот параметр задает размер буфера извлечения, используемого утилитой EXP. Если поделить значение параметра **BUFFER** на максимальный размер строки в этой таблице, можно определить, сколько строк за раз будет извлекать из таблицы утилита EXP. Чем больше размер буфера, тем выше производительность. Я пришел к выводу, что оптимальный размер буфера - около 100 строк. Учтите, что некоторые таблицы, в частности, содержащие столбцы типа **LONG** или большие двоичные объекты, считываются по одной строке, независимо от размера буфера. Нужно только проверить, достаточен ли размер буфера для размещения самого большого столбца.
COMPRESS	Y	Этот параметр **не задает сжатие экспортированных данных.** Он управляет генерацией конструкции **STORAGE** для экспортируемых объектов. Если оставить значение Y, конструкция хранения будет задавать для объектов начальный экстент, размер которого равен суммарному размеру их текущих экстентов. Т.е. утилита EXP будет генерировать оператор **CREATE** и с его помощью попытаться поместить весь объект в одном экстенте. Рекомендую устанавливать **compress** = **N** и использовать локально управляемые табличные пространства.

Имя параметра	*Стандартное значение*	*Назначение/Примечания*
ROWS		Указывает утилите EXP, следует экспортировать ли строки данных таблиц или только структуру. Я часто использую этот параметр со значением N для экспортирования структур.
FILESIZE		Если имеет положительное значение, файл **DMP,** создаваемый утилитой экспорта, устанавливается в максимальный размер. Используется при экспорте более двух гигабайт данных. Подробнее см. в разделе "Экспортирование больших объемов данных".
QUERY	нет	Позволяет связывать конструкцию **WHERE** с экспортируемыми таблицами. Конструкция **WHERE** будет применяться к строкам в ходе экспорта на уровне таблиц, при этом будут экспортироваться только строки, удовлетворяющие конструкции **WHERE.** Это позволяет экспортировать "срез" таблицы. Пример см. в разделе "Выделение подмножеств данных".
FULL	N	Если имеет значение Y, экспортируется вся база данных. При этом выбираются все пользователи, определения табличных пространств, системные привилегии и остальное содержимое базы данных.
OWNER	нет	Позволяет задать список схем для экспорта. Используется для клонирования схемы или "переименования" пользователя.
TABLES	нет	Позволяет задать список экспортируемых таблиц.
PARFILE	нет	Задает имя файла параметров, содержащего пары **parameter_name** = **values.** Может использоваться как альтернативный вариант заданию всех параметров в командной строке. Чаще всего используется для задания длинных списков экспортируемых таблиц или параметра QUERY.
CONSISTENT	N	Указывает, должно ли экспортирование выполняться в транзакции только для чтения. Это гарантирует согласованность различных таблиц. Как было описано в главе 3, каждый отдельный запрос выполняется как согласованный по чтению. Транзакция только для чтения (или с уровнем изолированности **SERIALIZABLE)** распространяет согласованность по чтению до уровня транзакции. Если экспортируются таблицы, связанные декларативным требованием целостности ссылок (RI - Referential Integrity) или вложенные таблицы и в дальнейшем планируется импортировать их вместе, рекомендуется использовать параметр **consistent** = Y. Это особенно важно, если велика вероятность изменения таблиц при экспортировании.

Имя параметра	*Стандартное значение*	*Назначение/Примечания*
TRANSPORT_TABLESPACE	N	Указывает, будет ли утилита EXP использоваться для экспортирования метаданных набора переносимых табличных пространств. Подробнее об этом см. в разделе "Перенос данных".
TABLESPACES	нет	Используется совместно с параметром **TRANSPORT_TABLESPACE,** чтобы задать список табличных пространств для переноса.

Параметры утилиты IMP

Вот какой результат выдает утилита IMP при передаче ей параметра HELP = Y:

```
C:\exp>imp help=y

Import: Release 8.1.6.0.0 - Production on Mon Mar 19 16:10:14 2001

(c) Copyright 1999 Oracle Corporation.  All rights reserved.

You can let Import prompt you for parameters by entering the IMP
command followed by your username/password:

      Example: IMP SCOTT/TIGER

Or, you can control how Import runs by entering the IMP command followed
by various arguments. To specify parameters, you use keywords:

      Format:   IMP KEYWORD=value or KEYWORD=(value1,value2,...,valueN)
      Example:  IMP SCOTT/TIGER IGNORE=Y TABLES=(EMP,DEPT)  FULL=N

                  or TABLES=(T1:P1,T1:P2), if T1 is partitioned table

USERID must be the first parameter on the command line.

Keyword Description  (Default)        Keyword       Description  (Default)
USERID  username/password             FULL          import entire file  (N)
FILE    input files  (EXPDAT.DMP)     TOUSER        list of usernames
SHOW    just list file contents  (N)  TABLES        list of table names
IGNORE  ignore create errors  (N)     RECORDLENGTH  length of IO record
GRANTS  import grants  (Y)            INCTYPE       incremental import type
INDEXES import indexes  (Y)           COMMIT        commit array insert  (N)
ROWS    import data rows  (Y)         PARFILE       parameter filename
LOG     log file of screen output     CONSTRAINTS   import constraints  (Y)
DESTROY  overwrite tablespace data file  (N)
INDEXFILE write table/index info to specified file
SKIP_UNUSABLE_INDEXES  skip maintenance of unusable indexes  (N)
ANALYZE  execute ANALYZE statements in dump file  (Y)
FEEDBACK display progress every x rows(0)
```

```
TOID_NOVALIDATE  skip validation of specified type ids
FILESIZE maximum size of each dump file
RECALCOLATE_STATISTICS recalculate statistics  (N)

The following keywords only apply to transportable tablespaces
TRANSPORT_TABLESPACE import transportable tablespace metadata  (N)
TABLESPACES tablespaces to be transported into database
DATAFILES datafiles to be transported into database
TTS_OWNERS users that own data in the transportable tablespace set

Import terminated successfully without warnings.
```

Давайте рассмотрим существенные параметры, не описанные в разделе, посвященном утилите EXP.

Имя параметра	*Стандартное значение*	*Назначение/Примечания*
SHOW	N	Если установлено значение Y, утилита импорта покажет свои потенциальные действия, не выполняя импортирование реально. Если задан параметр **SHOW = Y,** объекты не создаются и данные не добавляются.
IGNORE	N	Если установлено значение Y, IMP будет игнорировать **большинство** ошибок создания объектов. Пригодится, если объекты уже созданы в базе данных и IMP используется только для наполнения таблиц данными.
INDEXFILE	нет	Если этот параметр задан, IMP будет сбрасывать все операторы **CREATE INDEX** и множество других операторов ЯОД в указанный файл индексов (с комментариями в начальных строках, начинающихся с **REM).** Другие объекты из файла **DMP** не обрабатываются, создается только файл индексов.
FROMUSER	нет	С помощью этого параметра задают список пользователей, объекты которых надо импортировать из файла **DMP.** Можно использовать для восстановления одной схемы из файла экспорта всей базы данных.
TOUSER	нет	Если этот параметр указан, объекты пользователя, задаваемого параметром **FROMUSER,** импортируются в пользовательскую схему, имя которой является значением параметра **TOUSER.** Это позволяет "клонировать" пользовательскую схему.

Имя параметра	*Стандартное значение*	*Назначение/Примечания*
COMMIT	N	Указывает, должна ли утилита IMP фиксировать изменения после каждой множественной вставки. Количество вставляемых строк определяется параметром **BUFFER.** Обычно утилита IMP выполняет **COMMIT** после полной загрузки таблицы. Поскольку операторы вставки генерируют минимальный объем данных отката, при частом фиксировании замедляется вставка и увеличивается объем информации, записываемой в журналы повторного выполнения. Кроме того, продолжить работу IMP с места сбоя нельзя, поэтому я рекомендую оставлять для параметра значение N.
TTS_OWNERS	нет	При использовании вместе с параметром **TRANSPORTABLE_TABLESPACES** задает список владельцев объектов в переносимом табличном пространстве.

Экспортирование больших объемов данных

При использовании утилиты **EXP** для записи на устройство, поддерживающее произвольную адресацию (seeking), например, в обычные файлы, она ограничена максимальным размером генерируемого файла. Утилита **EXP** использует обычные библиотеки функций ОС, что на 32-битовых операционных системах ограничивает размер файла 2 Гбайтами. Я знаю четыре решения этой проблемы (хотя, вероятно, есть и другие).

Использование параметра FILESIZE

Этот параметр впервые появился в Oracle 8i. С помощью параметра **FILESIZE** можно установить максимальный размер (в байтах) файлов **DMP,** создаваемых в ходе экспортирования, и утилита EXP будет создавать столько файлов, сколько необходимо для экспорта всех данных. Например, чтобы экспортировать в набор файлов, размер каждого из которых не должен превосходить 500 Мбайт, можно использовать команду:

```
exp userid=tkyte/tkyte file = f1,f2,f3,f4,f5 filesize=00m owner = scott
```

В результате будут созданы DMP-файлы **f1.dmp, f2.dmp** и так далее, и размер каждого из них не будет превышать 500 Мбайт. Если общий объем экспортируемых данных будет менее 2 Гбайт, утилите **EXP** не придется создавать файл **f5.dmp.**

Проблема только в том, что невозможность заранее оценить объем экспортируемых данных делает процесс экспортирования интерактивным и трудно автоматизируемым. Рассмотрим сеанс экспорта, в котором около 2,3 Мбайт данных экспортируются в **DMP**-файлы размером 500 Кбайт:

```
C:\exp>exp userid=tkyte/tkyte tables=t file=(t1,t2,t3) filesize=500k

Export: Release 8.1.6.0.0 - Production on Mon Mar 19 14:54:12 2001
```

```
(c) Copyright 1999 Oracle Corporation.  All rights reserved.

Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option
JServer Release 8.1.6.0.0 - Production
Export done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR character
set
About to export specified tables via Conventional Path ...
. . exporting table                                          T

continuing export into file t2.DMP

continuing export into file t3.DMP

Export file: EXPDAT.DMP > t4

continuing export into file t4.DMP

Export file: EXPDAT.DMP > t5

continuing export into file t5.DMP
       21899 rows exported
Export terminated successfully without warnings.
```

Текст **"Export file: EXPDAT.DMP>"** является интерактивным приглашением. Использовав все имена файлов, заданные в командной строке — (t1, *tl,* t3), — утилита EXP начала запрашивать имя следующего файла. Если бы речь шла о пакетном сценарии, работающем поздно ночью, то утилита EXP просто ждала бы ответа на запрос или (в зависимости от обстоятельств вызова) аварийно завершила работу, не получив ответа. Этот способ может оказаться приемлемым во многих случаях. Если, предположим, известно, что объем экспорта не превысит 100 Гбайт (разумное предположение для базы данных объемом 50 Гбайт, например), можно задать для **FILESIZE** значение два гигабайта и сгенерировать список из 50 имен файлов в файле параметров **(PARFILE)** с помошью сценария. Затем просто указать **PARFILE = thatlist.par** вместо **FILE = (очень** длинный **список).**

Для импортирования этих данных надо вызвать утилиту **IMP** и передать ей список файлов **в порядке их создания.** Утилита IMP не проверят порядок файлов; выявив нарушение последовательности, она аварийно завершит работу. К счастью, можно указать больше файлов, чем фактически понадобится. Так что при использовании рекомендованного выше файла **PARFILE** в нем можно задавать и несуществующие файлы — утилита **IMP** на это не отреагирует. Вот пример:

```
C:\exp>imp userid=tkyte/tkyte full=y file=(t1,t2,t3,t4,t5,t6)

Import: Release 8.1.6.0.0 - Production on Mon Mar 19 15:49:24 2001

(c) Copyright 1999 Oracle Corporation.  All rights reserved.

Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option
JServer Release 8.1.6.0.0 - Production
```

```
Export file created by EXPORT:V08.01.06 via conventional path
import done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR character
set
IMP-00046: using FILESIZE value from export file of 512000
. inserting TKYTE's objects into TKYTE
. . importing table                            "T"        21899 rows
imported
Import terminated successfully with warnings.
```

Экспортирование по частям

Это способ решить проблему, просто избегая ее. Если имеется база данных объемом 10 Гбайт с 50 схемами приложений, причем объем данных в каждой схеме не превосходит 2 Гбайт, можно выполнять экспортирование на уровне пользователей. В результате будет экспортировано 50 файлов, каждый из которых содержит схему одного приложения.

Экспортирование в именованный канал

Это решение прекрасно работает в ОС UNIX. Я пока не нашел способа реализовать его в среде Windows. В данном случае с помощью команды **mknod** создается именованный канал. Именованный канал — это специальный файл, с помощью которого один процесс может записывать в канал данные, а процесс с другой стороны канала — читать их. Утилита EXP может записывать в каналы неограниченный объем данных, поскольку каналы не поддерживают абсолютную адресацию. Процесс, читающий данные из канала, может выполнять сжатие данных. Таким образом, можно одновременно экспортировать и сжимать данные. Если размер сжатого файла превышает 2 Гбайт, можно использовать утилиту split для разбиения его на меньшие части. Ниже представлен сценарий командного интерпретатора, реализующий этот подход в ОС UNIX. Этот сценарий также показывает, как импортировать сжатые и разбитые на части данные, поскольку сразу после экспортирования выполняется полное импортирование с параметром SHOW = Y для проверки целостности созданного DMP-файла:

```
#!/bin/ceh -f

# Установите эту переменную равной значению идентификатора пользователя, от
# имени которого выполняется экспорт. Я всегда использую учетные записи
# OPS$ (аутентифицируемые операционной системой) для всех заданий,
# выполняемых в пакетном режиме. В этом случае в файле сценария или
# результатах работы команды ps никогда не появляется пароль,
setenv UID /

# Это имя файла экспорта. Команда SPLIT будет использовать его для
# именования фрагментов сжатого DMP-файла.
setenv FN  exp.`date  +%j_%Y`.dmp

# Это имя используемого именованного канала,
setenv PIPE  /tmp/exp_tmp_ora8i.dmp
```

```
# Здесь я ограничиваю размер сжатых файлов до 500 Мбайт. Подойдет любое
# значение меньше 2 Гбайт,
setenv MAXSIZE 500m

# Задаем что экспортировать. По умолчанию экспортируется вся база данных,
setenv EXPORT_WHAT "full=y COMPRESS=n"


# Вот где размещаются файлы экспорта,
cd /nfs/atc-netappl/expbkup_ora8i

# Удаляем файлы, оставшиеся после предыдущего экспортирования,
rm expbkup.log export.test exp.*.dmp* $PIPE

# Создаем именованный канал,
mknod $PIPE p

# Записываем дату и время в журнальный файл,
date > expbkup.log

# Запускаем процесс gzip в фоновом режиме. Программа gzip будет считывать
# сжатые данные из именованного канала и передавать их утилите split. Из
# полученных данных утилита split будет создавать файлы размером 500 Мбайт,
# добавляя суффиксы .aa, .ab, .ac, .ad, ... к шаблону имени файла,
# заданному в переменной $FN.
( gzip < $PIPE ) | split -b $MAXSIZE - $FN. &

# Теперь начинаем экспортировать. Вызванная ранее программа gzip ожидает,
# пока в именованный канал начнут выдаваться данные.
exp userid=$UID buffer=20000000 file=$PIPE $EXPORT_WHAT >>& expbkup.log
date >> expbkup.log

# По завершении экспорта утилита IMP используется следующим образом. Имена
# файлов сортируются, после чего их содержимое с помощью команды cat
# направляется последовательно утилите gunzip. Результат направляется в
# именованный канал. Утилита IMP считывает данные из этого канала и выдает
# соответствующие результаты в стандартный поток ошибок. Конструкция >>& в
# командном интерпретаторе csh переправляет в указанный файл как
# стандартный выходной поток, так и стандартный поток ошибок.

date > export.test
cat 'echo $FN.* | sort | gunzip > $PIPE &
imp userid=$UID file=$PIPE show=y full=y >>& export.test
date >> export.test

# Удаляем именованный канал, он нам больше не нужен,
rm -f $PIPE
```

Если сервер работает в ОС UNIX, лично мне описанный выше подход кажется лучше, чем использование параметра FILESIZE = со списком имен файлов в командной строке, по двум причинам. Во-первых, он позволяет сжать данные перед записью на диск, а во-вторых, никогда не запрашивает у пользователя имена файлов.

Экспортирование на устройство, не поддерживающее абсолютную адресацию

Это решение тоже пригодно только для ОС UNIX. Можно экспортировать непосредственно на ленточное устройство, указав его имя в качестве файла экспорта. Например:

```
exp userid=tkyte/tkyte file=/dev/rmt/0 volsize=6000m full=y
```

Данные будут экспортироваться непосредственно на ленту с остановкой через каждые 6000 Мбайт для смены ленты (при необходимости).

Выделение подмножеств данных

В Oracle 8i появилась возможность использовать утилиту EXP для экспортирования отдельных строк таблицы. До этой версии утилита EXP работала по принципу "все или ничего". Сейчас можно использовать параметр **QUERY=** для задания конструкции WHERE, которая будет применяться к экспортируемой таблице. Следует отметить, что при использовании конструкции **WHERE** (параметра **QUERY)** непосредственный режим экспортирования (direct path mode) недоступен; если необходимо подмножество данных, экспортирование выполняется в обычном режиме (conventional path mode).

Способ задания параметра **QUERY** = зависит от операционной системы. Конструкция **WHERE** в общем случае содержит много специальных символов, таких как >, < и пробелы. Командные интерпретаторы в UNIX и Windows не "жалуют" подобные символы. Эти символы надо маскировать, а способ маскировки зависит от используемой операционной системы. Я предпочитаю всегда задавать параметр **QUERY** в файле **PARFILE.** Тогда можно использовать одни и те же командные строки, независимо от платформы.

Для демонстрации я создал таблицу **T** как **SELECT * FROM ALL_OBJECTS.** Я хочу экспортировать все строки, в которых столбец **object_id** имеет значение менее 5000. В ОС Windows для этого пришлось бы выполнить:

```
C:\exp>exp userid=tkyte/tkyte tables=t query="""where object_id < 5000"""
```

Обратите внимание, что в Windows необходимо указывать по три двойные кавычки с обеих сторон конструкции **WHERE.** Аналогичная команда в ОС UNIX имеет вид:

```
$ exp userid=tkyte/tkyte tables=t query=\"where object_id \< 5000\"
```

Однако если просто использовать файл параметров, **exp.par,** содержащий следующий аргумент:

```
query="where object_id < 5000"
```

то можно использовать в обеих системах одну и ту же команду:

```
exp userid=tkyte/tkyte tables=t parfile=exp.par
```

Я думаю, это намного проще, чем правильно замаскировать строки **QUERY** на разных платформах.

Перенос данных

Переносимое табличное пространство позволяет взять сформатированные файлы данных из одной базы данных и подключить их к другой. Вместо того чтобы выгружать данные в текстовый файл или файл **DMP,** а затем вставлять эти данные в другую базу, можно перенести табличное пространство. Это позволяет перемещать данные так же быстро, как и копировать файлы.

На переносимые табличные пространства налагается ряд ограничений.

- **Исходная и целевая базы данных должны работать на одной аппаратной платформе.** Нельзя, например, перенести файлы данных с Windows **NT** на **HP/UX.** Файл **DMP** можно копировать из одной ОС в другую, но файлы данных — нет; в отличие от файлов **DMP,** файлы данных — зависимы от ОС.
- **Исходная и целевая базы данных должны использовать один и тот же набор символов.** Нельзя, например, взять набор файлов из базы данных с набором символов WE8ISO8859P1 и подключить их к экземпляру, работающему в кодировке UTF8.
- **В целевой базе данных не должно быть табличного пространства с тем же именем.** Будет использоваться имя табличного пространства из исходной базы данных. Если в целевой базе данных уже есть табличное пространство с таким именем, сервер Oracle не сможет подключить одноименное.
- **Размеры блока в исходной и целевой базе данных должны совпадать.** Нельзя подключить файл из базы данных с размером блока 4 Кбайт к базе данных с размером блока 8 Кбайт.
- **Необходимо переносить самодостаточный набор объектов.** Например, нельзя переносить табличное пространство, содержащее индекс, не перенося при этом табличное пространство, содержащее индексируемую таблицу.
- **Нельзя переносить некоторые объекты.** Это материализованные представления, индексы по функциям, прикладные индексы (например, создаваемые компонентом interMedia), ссылки и очереди (advanced queues) с несколькими реципиентами.
- **В исходной базе данных переносимое табличное пространство должно быть временно переведено в режим READ ONLY.** Речь идет о периоде времени, достаточном для экспортирования метаданных табличного пространства и копирования файлов данных.
- **Нельзя переносить объекты, принадлежащие пользователю SYS.** Если в табличном пространстве имеются объекты, принадлежащие SYS, перенос завершится неудачно. Это означает, что такие объекты, как сегменты отката, табличное пространство SYSTEM и др., не могут быть перенесены (что логично, поскольку нет смысла транспортировать эти объекты).

В следующем примере показаны все шаги, необходимые для переноса табличного пространства. Чтобы было интереснее, я использую два табличных пространства. Начнем с организации табличных пространств, таблиц и создания нового пользователя:

```
SQL> create tablespace tts_ex1
  2  datafile 'c:\oracle\oradata\tkyte816\tts_ex1.dbf' size 1m
  3  extent management local uniform size 64k;
Tablespace created.

SQL> create tablespace tts_ex2
  2  datafile 'c:\oracle\oradata\tkyte816\tts_ex2.dbf size 1m
  3  extent management local uniform size 64k;
Tablespace created.

SQL> create user tts_user identified by tts_user
  2  default tablespace tts_ex1
  3  temporary tablespace temp;
User created.

SQL> grant dba to tts_user;
Grant succeeded.

SQL> connect tts_user/tts_user
Connected.

SQL> create table emp as select * from scott.emp;
Table created.

SQL> create table dept as select * from scott.dept;
Table created.

SQL> create index emp_idx on emp(empno) tablespace tts_ex2;
Index created.

SQL> create index dept_idx on dept(deptno) tablespace tts_ex2;
Index created.

SQL> select object_type, object_name,
  2                    decode(status, 'INVALID' ,'*','') status,
  3                     tablespace_name
  4  from user_objects a, user_segments b
  5  where a.object_name = b.segment_name (+)
  6  order by object_type, object_name
  7  /

OBJECT_TYPE    OBJECT_NAME          S   TABLESPACE_NAME

INDEX          DEPT_IDX                 TTS_EX2
               EMP_IDX                  TTS_EX2

TABLE          DEFT                     TTS_EX1
               EMP                      TTS_EX1
```

Перед экспортированием необходимо убедиться, что переносится самодостаточный набор объектов. Можно переносить таблицу без индексов, но нельзя переносить индексы без соответствующих таблиц. Ниже показана процедура проверки самодостаточности табличного пространства или набора табличных пространств:

```
SQL> exec sys.dbms_tts.transport_set_check('tts_exl', TRUE);
PL/SQL procedure successfully completed.

SQL> select * from sys.transport_set_violations;
no rows selected

SQL> exec sys.dbms_tts.transport_set_check('tts_ex2' , TRUE);
PL/SQL procedure successfully completed.

SQL> select * from sys.transport_set_violations;

VIOLATIONS

Index TTS_USER.EMP_IDX in tablespace TTS_EX2 points to table TTS_USER.BMP
in tablespace TTS_EX1
Index TTS_USER.DEPT_IDX in tablespace TTS_EX2 points to table TTS_USER.DEPT
in tablespace TTS_EX1

SQL> exec sys.dbms_tts.transport_set_check('tts_exl, tts_ex2', TRUE);
PL/SQL procedure successfully completed.

SQL> select * from sys.transport_set_violations;
no rows selected
```

Табличное пространство TTS_EX1 можно переносить, поскольку оно содержит только данные таблицы и является самодостаточным. Однако попытка перенести табличное пространство TTS_EX2 закончится неудачей, поскольку оно содержит индексы, но не таблицы, по которым эти индексы построены. Два табличных пространства, TTS_EX1 и TTS_EX2, можно перенести вместе, поскольку при этом переносятся и таблицы, и индексы.

Процедуру **SYS.DBMS_TTS** может выполнять любой администратор базы данных (администраторы обычно имеют привилегию **EXECUTE ANY PROCEDURE**) или любой пользователь, которому предоставлена роль **EXECUTE_CATALOG_ROLE**. В результате выполнения этой процедуры в динамически создаваемую таблицу записываются все ошибки, которые могут возникнуть при попытке переноса указанных табличных пространств. Теперь все готово для "отсоединения" или переноса соответствующих табличных пространств. Начнем с перевода их в режим READ ONLY:

```
SQL> alter tablespace tts_exl read only;
Tablespace altered.

SQL> alter tablespace tts_ex2 read only;
Tablespace altered.
```

Затем выполняется команда EXP:

```
SQL> host exp userid="""sys/change_on_install as sysdba"""
transport_tablespace=y
tablespaces=(tts_exl,tts_ex2)
Export: Release 8.1.6.0.0 - Production on Mon Mar 19 19:26:26 2001

(c) Copyright 1999 Oracle Corporation.  All rights reserved.

Production
With the Partitioning option
```

```
JServer Release 8.1.6.0.0 - Production
Export done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR character
set
Dote: table data (rows) will not be exported
About to export transportable tablespace metadata...
For tablespace TTS_EX1 . . .
. exporting cluster definitions
. exporting table definitions
. . exporting table                          EMP
. . exporting table                          DEPT
For tablespace TTS_EX2 . . .
. exporting cluster definitions
. exporting table definitions
. exporting referential integrity constraints
. exporting triggers
. end transportable tablespace metadata export
Export terminated successfully without warnings.
```

Обратите внимание на использование утроенных двойных кавычек для указания идентификатора пользователя в командной строке. В ОС UNIX придется маскировать еше и символ /. Чтобы избежать этого, можно дать указание утилите EXP запросить имя пользователя. Учтите также, что использована конструкция **AS SYSDBA.** Только пользователь **SYSDBA (internal)** может выполнять перенос в Oracle 8.1.6 и более новых версиях. В Oracle 8.1.5 было достаточно роли **DBA.** (Обратите внимание: в SQL*Plus эта команда должна вводиться одной строкой. В представленном выше примере строка перенесена.)

Теперь осталось только скопировать файлы данных в другое место. Это можно делать параллельно с экспортированием, чтобы сократить время, в течение которого для пространств установлен режим "только для чтения":

```
SQL> host XCOPY c:\orade\oradata\tkyte816\tts_ex7.dbf c:\temp
C:\orade\oradata\tkyte816\TTS_EXl.DBF
C:\oracle\oradata\tkyte816\TTS_EX2.DBF
2 File(s) copied

SQL> alter tablespace tts_ex1 read write;
Tablespace altered.

SQL> alter tablespace tts_ex2 read write;
Tablespace altered.
```

Табличные пространства снова доступны для чтения и записи. Теперь можно перенести эти файлы и присоединить их к другой базе данных:

```
C:\exp> imp file=expdat.drop userid="""sys/manager as sysdba"""
    transport_tablespace=y
    "datafiles=(c:\temp\tts_exl.dbf,c:\temp\tts_ex2.dbf)"

Import: Release 8.1.6.0.0 - Production on Mon Mar 19 19:26:39 2001

(c) Copyright 1999 Oracle Corporation.  All rights reserved.


Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 — Production
```

```
With the Partitioning option
JServer Release 8.1.6.0.0 - Production

Export file created by EXPORT:V08.01.06 via conventional path
About to import transportable tablespace(s) metadata...
import done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR character
set
. importing SYS's objects into SYS
. importing TTS_JSER's objects into TTS_USER
   . importing table                              "EMP"
   . importing table                             "DEPT"
Import terminated successfully without warnings.

SQL> update emp set ename=lower(ename);
update emp set ename=lower(ename)
       *
ERROR at line 1:
ORA-00372: file 9 cannot be modified at this time
ORA-01110: data file 9: 'C:\TEMP\TTS_EX1.DBF'

SQL> alter tablespace tts_ex1 read write;
Tablespace altered.

SQL> alter tablespace tts_ex2 read write;
Tablespace altered.

SQL>  update emp set ename=lower(ename);
14 rows updated.
```

Вот и все, файлы присоединены к базе данных. Последний шаг показывает, что они присоединяются в режиме READ ONLY (это логично, поскольку при переносе для них был установлен режим "только для чтения"). После присоединения может потребоваться изменить режим доступа к ним. Если вы хотите проверить эту процедуру в одной базе данных, можете выполнить следующие или аналогичные команды в базе данных после перевода табличных пространств обратно в режим READ WRITE, но перед импортированием:

```
SQL> drop tablespace tts_ex1 including contents;
Tablespace dropped.

SQL> drop tablespace tts_ex2 including contents;
Tablespace dropped.

SQL> host erase c:\oracle\oradata\tkyte816\tts_ex7.dbf
```

Именно так я "сбрасываю" базу данных для тестирования. Я переношу первоначальную тестовую базу данных перед тестированием, а когда необходимо сбросить изменения, удаляю существующие табличные пространства и снова присоединяю первоначальную базу данных.

Переносимые табличные пространства можно также использовать для ручного восстановления табличного пространства на определенный момент времени. Предположим, вы случайно удалили таблицу. Можно восстановить табличные пространства SYSTEM, ROLLBACK и затронутое табличное пространство на другой машине. Эта мини-база дан-

ных восстанавливается на момент времени, непосредственно предшествующий ошибочному удалению таблицы. Теперь можно перенести табличное пространство, содержащее эту таблицу, в другую базу данных и снова присоединить его. Такой же эффект дает диспетчер восстановления **(RMAN)** при восстановлении табличного пространства по состоянию на определенный момент времени. Если вы не используете **RMAN,** можете выполнить эту операцию самостоятельно.

Еще одна возможность применения переноса связана с совместным использованием больших объемов только читаемых (или минимально изменяющихся) данных двумя экземплярами на одной машине. Можно создать большое табличное пространство, перевести его в режим "только для чтения", экспортировать метаданные, а затем импортировать в другой экземпляр. Получаем две базы данных с доступом только для чтения к одному и тому же набору файлов. Если в дальнейшем понадобится изменить информацию, необходимо будет выполнить следующие действия:

- удалить табличное пространство (включая содержимое) в базе данных, к которой присоединялись файлы данных;
- перевести табличное пространство в режим чтения и записи в исходной базе данных;
- выполнить необходимые изменения;
- перевести табличное пространство в режим "только для чтения";
- экспортировать метаданные и снова импортировать их в другую базу данных.

Если табличное пространство используется несколькими базами данных, оно должно быть доступно в режиме "только для чтения".

Получение операторов ЯОД

Утилиту EXP можно использовать для получения большей части операторов ЯОД, формирующих базу данных. Это уже было продемонстрировано в главе 6, где я использовал EXP и **IMP** для получения детального оператора **CREATE TABLE.**

Есть два способа получить операторы ЯОД: **SHOW=Y** и **INDEXFILE = имя_файла. Я** всегда рекомендую использовать параметр **INDEXFILE,** а не **SHOW=Y.** Последний параметр предназначен для отображения действий утилиты EXP при реальном экспортировании. Формат выдаваемых результатов такой, что в исходном виде они бесполезны: зачастую в самых неподходящих местах операторы ЯОД переносятся на следующую строку и вставляются двойные кавычки. Кроме того, нет четкого разделения команд. Параметр **SHOW** = Y хорошо подходит как последнее средство восстановления хотя бы части операторов ЯОД, если другие способы недоступны. Ниже мы сравним результаты использования этих параметров, и вы поймете, почему надо использовать **INDEXFILE.**

Задав параметр **INDEXFILE,** можно воссоздать в файле сценария большую часть операторов ЯОД, формирующих схему. Например, если начать так:

```
tkyte@TKYTE816> create table t1  (x int primary key, y int) ;
Table created.
```

```
tkyte@TKYTE816> create table t2 (coll int references tl, col2 int check
(col2>0));
Table created.

tkyte@TKYTE816> create index t2_idx on t2(col2,coll);
Index created.

tkyte@TKYTE816> create trigger t2_trigger before insert or update of coll,
                                          col2 on t2 for each row
  2  begin
  3       if (:new.coll < :new.col2) then
  4           raise_application_error(-20001,
  5                    'Invalid Operation Coll cannot be less then Col2') ;
  6     end if;
  7 end;
  8  /
Trigger created.

tkyte@TKYTE816> create view v
  2  as
  3   select tl.y tl_y, t2.col2 t2_col2 from tl, t2 where tl.x = t2.coll
  4  /
View created.
```

То можно затем запустить EXP и **IMP:**

```
C:\>exp userid=tkyte/tkyte owner=tkyte
C:\>imp userid=tkyte/tkyte full=y indexfile=tkyte.sql
```

В файле **tkyte.sql** будет:

```
REM  CREATE TABLE "TKYTE"."T1" ("X" NUMBER(*,0), "Y" NUMBER(*,0)) PCTFREE
REM  10 PCTUSED 40 INITRANS 1 MAXTRANS 255 LOGGING STORAGE(INITIAL 524288)
REM  TABLESPACE "DATA" ;
REM  ... 0 rows
REM  ALTER TABLE "TKYTE". "T1" ADD PRIMARY KEY ("X") USING INDEX PCTFREE 10
REM  INITRANS 2 MAXTRANS 255 STORAGE (INITIAL 524288) TABLESPACE "DATA"
REM  ENABLE ;
REM  CREATE TABLE "TKYTE"."T2" ("COL1" NUMBER(*,0), "COL2" NUMBER(*,0))
REM  PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 LOGGING STORAGE (INITIAL
REM  524288) TABLESPACE "DATA" ;
REM  ... 0 rows
CONNECT TKYTE;
CREATE INDEX "TKYTE". "T2_IDX" ON "T2" ("COL2" , "COL1" ) PCTFREE 10
INITRANS 2 MAXTRANS 255 STORAGE(INITIAL 524288) TABLESPACE "DATA" LOGGING ;
REM  ALTER TABLE "TKYTE". "T2" ADD CHECK (col2>0) ENABLE ;
REM  ALTER TABLE "TKYTE". "T2" ADD FOREIGN KEY ("COL1") REFERENCES "T1"
REM  ("X") ENABLE ;
```

Удалив операторы **REM,** получим операторы ЯОД для создания объектов, занимающих место на диске, но не для триггера или представления (не будет также процедур, пакетов и т.д.). Утилита EXP экспортирует эти объекты, но утилита **IMP** их не показывает при задании параметра **INDEXFILE.** Один из способов заставить утилиту **IMP** показать их — задать параметр **SHOW:**

```
C:\ImpExp> imp userid=tkyte/tkyte show=y full=y
Import: Release 8.1.6.0.0 - Production on Mon Apr 23 15:48:43 2001
(c) Copyright 1999 Oracle Corporation.  All rights reserved.
Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 — Production
With the Partitioning option
JServer Release 8.1.6.0.0 — Production

Export file created by EXPORT:V08.01.06 via conventional path
import done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR character set
. importing TKYTE's objects into TKYTE
 "CREATE TABLE "T1" ("X" NUMBER(*,0), "Y" NUMBER(*,0))  PCTFREE 10 PCTUSED 40"
 " INITRANS 1 MAXTRANS 255 LOGGING STORAGE(INITIAL 524288) TABLESPACE "DATA""
. . skipping table "T1"

 "CREATE TABLE "T2" ("COL1" NUMBER(*,0), "COL2" NUMBER(*,0))  PCTFREE 10 PCTU"
 "SED 40 INITRANS 1 MAXTRANS 255 LOGGING STORAGE(INITIAL 524288) TABLESPACE ""
 "DATA""
. . skipping table "T2"

 "CREATE INDEX "T2_IDX" ON "T2" ("COL2" , "COL1" ) PCTFREE 10 INITRANS 2 MAX"
 "TRANS 255 STORAGE(INITIAL 524288) TABLESPACE "DATA" LOGGING"
 "CREATE FORCE VIEW "TKYTE"."V"                          ("T1_Y","T2_COL2")  "
 "AS "
 "select t1.y t1_y, t2.col2 t2_col2 from t1, t2 where t1.x = t2.col1"
"CREATE TRIGGER "TKYTE".t2_trigger before insert or update of col1, col2 on "
 "t2 for each row"


 "begin"
 "     if (:new.col1 < :new.col2) then"
 "          raise_application_error(-20001,'Invalid Operation Col1 cannot be le"
 "ss then Col2');"
 "     end if;"
 "end;"
 "ALTER TRIGGER "T2_TRIGGER"  ENABLE"
Import terminated successfully without warnings.
```

Обратите внимание: результат абсолютно не подходит для обычного использования. Например, рассмотрим фрагмент:

```
"CREATE TABLE "T2" ("COL1" NUMBER(*,0), "COL2" NUMBER(*,0)) PCTFREE 10 PCTU"
 "SED 40 INITRANS 1 MAXTRANS 255 LOGGING STORAGE(INITIAL 524288)
    TABLESPACE ""
```

Утилита IMP разрывает строки в произвольных местах — слово PCTUSED разбито на части. Кроме того, каждая строка начинается и завершается двойной кавычкой. Удаление этих кавычек еще не сделает файл сценария применимым, поскольку команды разбиты на строки в произвольных местах. Более того, сам исходный код также "поврежден":

```
"   if ( :new.col1 < :new.col2 ) then"
"       raise_application_error(-20001,'Invalid Operation Col1 cannot be le"
"ss then Col2');"
"    end if;"
```

Утилита IMP вставила символ перевода строки прямо в середину строки кода, в строковый литерал. Наконец, сами команды никак не разделены:

```
"CREATE INDEX "T2_IDX" ON "T2" ("COL2" , "COL1") PCTFREE 10 INITRANS 2 MAX"
"TRANS 255 STORAGE(INITIAL 524288) TABLESPACE "DATA" LOGGING"
"CREATE FORCE VIEW "TKYTE"."V"                    ("T1_Y","T2_COL2")  "
"AS "
"select t1.y t1_y, t2.col2 t2_col2 from t1, t2 where t1.x = t2.col1"
"CREATE TRIGGER "TKYTE".t2_trigger before insert or update of col1, col2 on "
"t2 for each row"
```

Команда CREATE INDEX "плавно" переходит в CREATE VIEW, а та, в свою очередь, — в CREATE TRIGGER и так далее (надеюсь, вы поняли!). Чтобы использовать этот файл, его надо тщательно редактировать. Но, мы получили недостающее: хотя формат и не подходит для непосредственного выполнения, все можно восстановить. Такой файл может пригодиться в том случае, если вы случайно удалили весь созданный за последний месяц код и хотите его восстановить. Вот почему я регулярно экспортирую свою базу данных дважды в неделю и применяю к результатам **IMP...SHOW = Y** (как было показано в разделе "Экспортирование больших объемов данных"). Неоднократно мне удавалось восстановить для клиентов достаточно актуальную копию кода по результатам выполнения этой команды. Это избавляло от необходимости восстанавливать для получения кода всю базу данных на определенный момент времени. (Это был еще один альтернативный вариант: данные в базе никогда не теряются!)

Преодоление ограничений с помощью сценариев

Если мне необходимо перенести из одной схемы в другую код PL/SQL, я предпочитаю использовать сценарии. У меня есть сценарий для получения исходных текстов пакета, процедуры или функции. Другой сценарий предназначен для извлечения представлений. Еще один позволяет получить исходный текст триггера. Перенос таких объектов — не лучший способ использования утилит EXP/IMP. Переносите с помощью EXP/IMP таблицу — и все будет отлично. Попробуйте с помощью EXP/IMP извлечь несколько определений таких объектов, и все станет менее радужным.

Поскольку эти сценарии весьма полезны, я представил их здесь, в главе, посвященной EXP. Возможно, читая этот раздел, вы пытались найти способ получения исходного кода с помощью утилиты IMP. Теперь вы знаете, что IMP не позволяет получить его в удобочитаемом виде.

Итак, вот сценарий, извлекающий исходный код любого пакета (в том числе тела пакета), функции или процедуры и записывающий его в одноименный файл SQL. Если выполнить команду SQL> @getcode my_procedure, этот сценарий создаст файл my_procedure.sql, содержащий соответствующую подпрограмму PL/SQL:

```
REM getcode.sql — выбирает любую процедуру, функцию или пакет
set feedback off
set heading off
set termout off
set linesize 1000
```

```
set trimspool on
set verify off
spool &1..sql
prompt set define off
select decode(type||'-'||to_char(line,'fm99999'),
                  'PACKAGE BODY-1', '/'||chr(10),
                   null) ||
       decode(line,1,'create or replace ', '') ||
       text text
  from user_source
 where name = upper('&&1')
 order by type, line;
prompt /
prompt set define on
spool off
set feedback on
set heading on
set termout on
set linesize 100
```

Для тех, кто хотел бы получить **весь** код из схемы, я предлагаю сценарий **getallcode.sql.** Сначала он создаст по файлу для каждого хранимого объекта PL/SQL в текущем каталоге, а затем — сценарий **getallcode_INSTALL,** который будет автоматически устанавливать весь этот код в другой схеме:

```
set termout off
set heading off
set feedback off
set linesize 50
spool xtmpx.sql
select '@getcode ' || object name
from user_objects
where object_type in ('PROCEDURE', 'FUNCTION', 'PACKAGE')
/
spool off
spool getallcode_INSTALL.sql
select '@' || object_name
from user_objects
where object_type in ('PROCEDURE' , 'FUNCTION', 'PACKAGE')
/
spool off
set heading on
set feedback on
set linesize 130
set termout on
@xtmpx.sql
```

Следующий сценарий позволяет выбрать код одного представления. Если выполнить команду **SQL> @getaview view_name,** он создаст в текущем каталоге файл **view_name.sql**, содержащий оператор **CREATE VIEW:**

```
REM getaview.sql
set heading off
set long 99999999
set feedback off
set linesize 1000
set trimspool on
set verify off
set termout off
set embedded on

column column_name format A1000
column text format A1000

spool &1..sql
prompt create or replace view 41 (
select decode(column_id,1,'',',') || column_name column_name
   from user_tab_colunms
 where table_name = upper('&1')
 order by column_id
/
prompt ) as
select text
   from user_views
 where view_name = upper('&1')
/
prompt /
spool off

set termout on
set heading on
set feedback on
set verify on
```

Конечно, если необходимо получить **все** представления, можно использовать CL
рий **getallviews:**

```
set heading off
set feedback off
set linesize 1000
set trimspool on
set verify off
set termout off
set embedded on

spool tmp.sql
select '@getaview ' || view_name
from user_views
/
spool off

set termout on
set heading on
set feedback on
set verify on
@tmp
```

Наконец, имеется сценарий **gettrig.sql**. Он обрабатывает не все триггеры. Например, я не извлекаю конструкцию referencing OLD as ..., поскольку никогда ее не использую. Идея — та же, что и в представленных ранее сценариях; этот сценарий очень легко изменить, если необходимо использовать корреляционные имена:

```
set echo off
set verify off
set feedback off
set termout off
set heading off
set pagesize 0
set long 99999999
spool &1..sql

select
'create or replace trigger " ' ||
          trigger_name || '"' || chr(10)||
 decode(substr(trigger_type, 1, 1),
           'A', 'AFTER', 'B', 'BEFORE', 'I', 'INSTEAD OF') ||
                chr(10) ||
 triggering_event || chr(10) ||
 'ON "' || table_owner || '"."' ||
        table_name || "" || chr(10) ||
 decode(instr( trigger_type, 'EACH ROW'), 0, null,
              'FOR EACH ROW) || chr(10),
 trigger_body
from user_triggers
where trigger_name = upper('&1')
/
prompt /

spool off
set verify on
set feedback on
set termout on
set heading on
```

Итак, мы показали, как с помощью утилит EXP/IMP получать операторы ЯОД для таблиц и индексов, занимающих определенное место на диске. Для объектов, много места не занимающих, в том числе триггеров, хранимых процедур, представлений, последовательностей, синонимов и т.д., больше подходят простые сценарии SQL*Plus. Утилиту **IMP** с параметром **SHOW=Y** можно использовать как последнее средство спасения, но если получать код требуется постоянно, лучше использовать сценарий.

Резервное копирование и восстановление

Утилиты EXP и IMP не следует использовать для резервного копирования и восстановления. Реальные резервные копии позволяет создавать только утилита RMAN и операционная система. Утилиты EXP/IMP не стоит использовать как средства резервного копирования по следующим причинам.

- Они, в лучшем случае, позволяют восстановить базу данных на определенный момент времени. При задании параметра CONSISTENT = Y можно получить моментальный снимок базы данных (помните, однако, что вероятность получить сообщение об ошибке **ORA-01555 snapshot too old** тем больше, чем длиннее транзакция), но не более того. Повторяю: это — снимок на определенный момент времени. Если использовать результат такого экспорта для восстановления, будут потеряны **все** изменения, произошедшие после запуска EXP. Кроме того, к результатам работы **IMP** нельзя применить архивные журналы повторного выполнения.
- Восстановление базы данных сколько-нибудь существенного размера с помощью утилиты **IMP** происходит медленно; будут вставляться все данные (через SQL-машину, с генерацией соответствующих объемов данных отката и повторного выполнения), все индексы придется перестраивать, все ограничения — проверять, весь код — компилировать и так далее. То, что при реальном восстановлении занимает минуты, потребует многих часов и даже дней при использовании утилиты **IMP.**
- Инкрементное экспортирование и импортирование с помощью EXP/IMP скоро перестанет поддерживаться. Использование параметра **INCTYPE** = будет запрещено. Вот вам цитата из документации Oracle: "Важно: Возможности инкрементного, кумулятивного и полного экспортирования — избыточны и будут убраны в следующих версиях. Необходимо уже сейчас переходить к использованию для резервного копирования баз данных диспетчера резервного копирования и восстановления Oracle". Подробнее см. в руководстве *"Oracle8i Operating System Backup and Recovery Guide".*

Означает ли это, что утилиты EXP/IMP становятся бесполезными в общесистемной стратегии резервного копирования и восстановления? Я, например, считаю, что они **могут** играть важную роль в общесистемной стратегии резервного копирования и восстановления. Производственная база данных должна работать в режиме архивирования журналов, чтобы обеспечить возможность восстановления "на момент времени" и восстановления носителей (т.е. восстановления после сбоя диска). Это принципиально важно, и альтернативы этому нет. Кроме того, определенное профилактическое выявление сбоев тоже важно в продуманной стратегии резервного копирования и восстановления. Именно для этого я и использую утилиту EXP, как уже упоминал ранее. При этом полностью проверяется словарь данных, используются практически все его индексы и объекты, что гарантирует целостность. В процессе экспортирования прочитываются все данные таблиц, что позволяет проверить их доступность (если индекс будет поврежден, его легко восстановить, поэтому я не беспокоюсь об их тестировании). Полученный файл **DMP** может также пригодиться для извлечения потерянных фрагментов кода или случайно удаленной таблицы, что во многих случаях позволяет избежать восстановления на определенный момент времени.

Есть и другие средства, такие как **DBV** (средство проверки базы данных), которые также можно периодически применять к файлам данных, чтобы обеспечить физическую целостность данных, в том числе индексных структур, не обрабатываемых утилитой **EXP.**

Утилиты IMP/EXP (уже) не являются средствами реорганизации

В прошлом утилиты экспорта и импорта использовались в основном в этих целях. Для того чтобы дефрагментировать табличное пространство, администратору базы данных приходилось сначала экспортировать набор объектов, затем удалить их и после этого импортировать. Администраторы базы данных тратили много времени на регулярное выполнение такой процедуры. Хотя на самом деле не требовалось делать это больше одного раза, а в большинстве случаев — вообще не нужно. Одноразового выполнения этой процедуры было достаточно, чтобы продумать более эффективную организацию хранения, позволяющую избежать фрагментации. В большинстве случаев, однако, никто ничего не менял, и история неизбежно повторялась. В некоторых случаях это делали, поскольку слышали, что "это необходимо", хотя в их случае этого можно было избежать.

Более того, подобное использование утилит EXP/IMP было небезопасно. Вы берете все данные из базы данных, удаляете их, а затем снова вставляете. Определенный промежуток времени данные не защищаются СУБД. Есть опасность, что утилита IMP не сработает (это ведь просто программа). Есть также опасность изменения данных при экспортировании (эти изменения в файл экспорта не попадут) и потери этих изменений. Можно потерять права доступа к объектам и т.д. Все это требует серьезного планирования и приостановки работы на длительное время.

В Oracle8i нет необходимости использовать утилиты EXP/IMP для реорганизации данных. Если вы действительно думаете, что это нужно (а я свято верю, что делать это надо не более **одного** раза для исправления неудачной реализации), то можно использовать оператор **ALTER TABLE MOVE** для переноса таблиц из одного табличного пространства в другое. В ходе такой реорганизации таблица будет доступна для запросов, но не для изменений. Сразу же после переноса индексы становятся недействительными и надо их пересоздать, так что на это время производительность запросов снизится. Но время простоя будет **существенно** меньше, чем при использовании утилит EXP/IMP, причем ни одной из перечисленных выше проблем не возникает; данные все время защищены СУБД, нет угрозы потери или изменения данных, процесс не затрагивает привилегии и т.д.

В заключение еще раз подчеркну: дни EXP/IMP как средства реорганизации данных сочтены. Даже не пытайтесь использовать их для этой цели.

Импортирование в другие структуры

Такая необходимость возникает достаточно часто. Как, например, файл экспорта с данными импортировать в отличную структуру? Я часто сталкивался с такой пробле-

мой, когда данные из версии 1 некоего программного пакета требовалось загрузить в базу данных, где установлена версия 2 (или наоборот). Ответ — да, но для этого потребуются определенные усилия. Рассмотрим три случая:

- в таблицу добавлен столбец (никаких дополнительных усилий не надо — сервер Oracle поместит в столбец пустые или заданные стандартные значения);
- из таблицы удален столбец (кое-что придется сделать);
- изменен тип данных столбца (опять-таки, придется кое-что сделать).

В случае появления дополнительного столбца делать ничего не надо. Сервер Oracle выполнит вставку в таблицу как обычно, используя пустые значения или указанное при добавлении столбца стандартное значение. При удалении или изменении столбцов придется импортировать данные в представление, используя триггер INSTEAD OF для их сопоставления. Учтите, что использование триггера INSTEAD OF приводит к дополнительным затратам ресурсов; загружать с его помощью можно небольшие объемы данных, но десятки миллионов строк — не стоит. Рассмотрим пример со следующими таблицами:

```
tkyte@TKYTE816> create table added_a_column (x int) ;
Table created.

tkyte@TKYTE816> create table dropped_a_column (x int, y int);
Table created.

tkyte@TKYTE816> create table modified_a_column(x int, y int);
Table created.

tkyte@TKYTE816> insert into added_a_column values (1) ;
1 row created.

tkyte@TKYTE816> insert into dropped_a_column values (1, 1) ;
1 row created.

tkyte@TKYTE816> insert into modified_a_column values (1, 1) ;
1 row created.

tkyte@TKYTE816> commit;
Commit complete.
```

Начнем с экспортирования этих трех таблиц (команда для этого должна вводиться в одну строку, иначе будет экспортирована вся схема):

```
tkyte@TKYTE816> host exp userid=tkyte/tkyte

tables=(added_a_column,dropped_a_column,modified_a_column)

Export: Release 8.1.6.0.0 - Production on Tue Mar 20 09:02:34 2001
(c) Copyright 1999 Oracle Corporation.  All rights reserved.

Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option
JServer Release 8.1.6.0.0 - Production
Export done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR character set
```

```
About to export specified tables via Conventional Path ...
. . exporting table                ADDED_A_COLUMN      1 rows exported
. . exporting table              DROPPED_A_COLUMN      1 rows exported
. . exporting table             MODIFIED_A_COLOMN      1 rows exported
Export terminated successfully without warnings.
```

Итак, создан тестовый пример. Мы экспортировали три таблицы "как они есть". Теперь давайте их изменим:

```
tkyte@TKYTE816> alter table added_a_colunn add (y int) ;
Table altered.

tkyte@TKYTE816> alter table dropped_a_column drop column y;
Table altered.

tkyte@TKYTE816> delete from modified_a_column;
1 row deleted.

tkyte@TKYTE816> alter table modified_a_column modify y date;
Table altered.
```

Теперь, если попытаться импортировать данные, с таблицей **ADDED_A_COLUMN** все получится, а вот с остальными — нет:

```
tkyte@TKYTE816> host imp userid=tkyte/tkyte full=y ignore=y

Import: Release 8.1.6.0.0 - Production on Tue Mar 20 09:02:34 2001
(c) Copyright 1999 Oracle Corporation.  All rights reserved.

Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option
JServer Release 8.1.6.0.0 - Production

Export file created by EXPORT:V08.01.06 via conventional path
import done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR character
set
. importing TKYTE's objects into TKYTE
. . importing table                "ADDED_A_COLUMN"       1 rows imported
. . importing table              "DROPPED_A_COLUMN"
IMP-00058: ORACLE error 904 encountered
ORA-00904: invalid column name
. . importing table                "MODIFIED_A_COLUMN"
IMP-00058: ORACLE error 932 encountered
ORA-00932: inconsistent datatypes
Import terminated successfully with warnings.
```

Следующий шаг — создать представления в базе данных, аналогичные исходным таблицам. Для этого:

- переименуем таблицы на время импортирования;
- создадим представления, выбирающие константы нужного типа; например, SELECT 1 - для чисел, SELECT SYSDATE - для дат, SELECT RPAD('*', 30, '*') - для VARCHAR2(30) и так далее;

- создадим триггер INSTEAD OF, который будет выполнять все необходимые действия (преобразование/сопоставление данных).

Вот код для этого:

```
tkyte@TKYTE816> rename modified_a_column to modified_a_column_TEMP;
Table renamed.

tkyte@TKYTE816> create or replace view modified_a_column
  2  as
  3  select 1 x, 1 y from modified_a_column_TEMP;
View created.

tkyte@TKYTE816> create or replace trigger modified_a_column_IOI
  2  instead of insert on modified_a_column
  3  begin
  4            insert into modified_a_column_TEMP
  5            (x, y)
  6            values
  7           (:new.x, to_date('01012001','ddmmyyyy')+:new.y);
  8  end;
  9  /
Trigger created.
```

Здесь мы преобразуем данные типа NUMBER, хранящиеся в столбце Y, в смещение от 1 января 2001 года. Выполните необходимые преобразования: из STRING — в DATE, из DATE — в NUMBER, из NUMBER — в STRING и так далее. Теперь позаботимся об удаленном столбце:

```
tkyte@TKYTE816> rename dropped_a_column to dropped_a_column_TEMP;
Table renamed.

tkyte@TKYTE816> create or replace view dropped_a_column
  2  as
  3  select 1 x, 1 y from dropped a_column_TEMP;
View created.

tkyte@TKYTE816> create or replace trigger dropped_a_column_IOI
  2  instead of insert on dropped_a_column
  3  begin
  4            insert into dropped_a_column_TEMP
  5            (x)
  6            values
  7            (:new.x);
  8  end;
  9  /
Trigger created.
```

Здесь мы избавились от столбца :new.y. Мы ничего с ним не делаем, просто игнорируем. Он должен быть в представлении, куда бы утилита IMP могла вставлять данные. Теперь все готово для повторного импортирования:

```
tkyte@TKYTE816> host imp userid=tkyte/tkyte full=y ignore=y

Import: Release 8.1.6.0.0 - Production on Tue Mar 20 09:21:41 2001
(c) Copyright 1999 Oracle Corporation.  All rights reserved.
```

```
Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option
JServer Release 8.1.6.0.0 - Production

Export file created by EXPORT:V08.01.06 via conventional path
import done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR character
set
. importing TKYTE's objects into TKYTE
. .. importing table           "ADDED_A_COLOMN"      1 rows imported
. . importing table          "DROPPED_A_COLDMN"      1 rows imported
. . importing table         "MODIFIKD_A_COLUMN"      1 rows imported
Import terminated successfully without warnings.
```

Импортирование выполняется без ошибок. Теперь необходимо удалить представления и переименовать таблицы:

```
tkyte@TKYTE816> drop view modified_a_column;
View dropped.

tkyte@TKYTE816> drop view dropped_a_column;
view dropped.

tkyte@TKYTE816> rename dropped_a_column_TEMP to dropped_a_column;
Table renamed.

tkyte@TKYTE816> rename modified_a_column_TEMP to modified_a_column;
Table renamed.
```

Анализируя данные, мы увидим следующее:

- три строки в таблице added_a_column — одна исходная и две импортированные;
- две строки в таблице dropped_a_column — результат исходной вставки и одного успешного импорта;
- одна строка в таблице modified_a_column, поскольку перед изменением типа столбца пришлось все строки из этой таблицы удалить.

А вот что мы получаем:

```
tkyte@TKYTE816> select * from added_a_column;

         X         Y
---------- ---------
         1
         1
         1

tkyte@TKYTE816> select * from dropped_a_column;

         X
 ----------
         1
         1

tkyte@TKYTE816> select * from modified_a_column;

         X Y

         1 02-JAN-01
```

Непосредственный экспорт

Непосредственный экспорт нельзя считать операцией, обратной непосредственной загрузке с помощью **SQLLDR** (см. главу 9). При непосредственном экспорте данные не читаются напрямую из файлов данных перед записью в DAT-файлы. Экспортирование в непосредственном режиме позволяет миновать буфер обработки SQL (избежать обработки конструкции **WHERE,** форматирования столбцов и т.д.). Девяносто процентов пути, проходимого данными, — те же. Утилита EXP так же считывает блоки в буферный кэш, обеспечивает согласованность по чтению и т.д.

Тем не менее ускорение при использовании непосредственного экспорта может оказаться существенным. Сокращение обработки всего на десять процентов оборачивается значительным сокращением времени выполнения. Например, я только что экспортировал около 100 Мбайт данных (1,2 миллиона записей). Непосредственный экспорт занял около минуты. Обычное же экспортирование продолжалось три минуты. К сожалению, соответствующего "непосредственного импорта" нет. При импортировании для вставки данных в таблицы используются обычные операторы SQL. Для высокопроизводительной загрузки данных по-прежнему приходится использовать **SQLLDR.**

Следует заметить, что в режиме непосредственного экспорта нельзя использовать параметр **QUERY** = для выбора подмножества строк. В этом есть смысл, если учесть, что **DIRECT = Y** — это просто способ обойти буфер обработки SQL, а ведь именно в нем обычно и обрабатывается конструкция **WHERE.**

Проблемы и ошибки

В этом разделе я хочу описать ряд проблем, с которыми часто сталкиваются при использовании утилит EXP/IMP. Мы рассмотрим:

- использование утилит EXP/IMP для "клонирования" схемы;
- использование утилит **EXP/IMP** в различных версиях Oracle;
- "исчезновение" индексов;
- влияние требований, имена которых сгенерированны системой;
- проблемы, связанные с поддержкой национальных языков (National Language Support- NLS);
- проблемы, возникающие при использовании объектов, ссылающихся на имена нескольких табличных пространств, например таблиц со столбцами типа больших двоичных объектов.

Клонирование

Это стандартное использование утилит EXP/IMP; для этого нужно скопировать всю схему приложения. Необходимо скопировать все таблицы, триггеры, представления, процедуры и т.д. Как правило, это замечательно получается с помощью следующих команд:

```
Exp userid=tkyte/tkyte owner=old_user
Imp userid=tkyte/tkyte fromuser=old_user touser=new_user
```

Неприятности, однако, возникают, когда в копируемой схеме приложения используются ссылки на ее же объекты, уточненные именем схемы. Я имею в виду, что в пользовательской схеме A имеется такой, например, код:

```
create trigger MY_trigger
before insert on A.table name
begin

end;
/
```

Здесь **явно** создается триггер для таблицы **A.TABLE_NAME.** Если экспортировать такой триггер и импортировать его в другую схему, он останется связанным с таблицей **A.TABLE_NAME,** а не с таблицей по имени **TABLE_NAME в** новой схеме. Утилиты EXP/IMP, однако, работают в этом случае по-разному:

```
tkyte@TKYTE816> create table tl
  2   (x int primary key);
Table created.

tkyte@TKYTE816> create table t4  (y int references TKYTE.tl);
Table created.

tkyte@TKYTE816> create trigger t2_trigger
  2  before insert on TKYTE.ti
  3  begin
  4     null;
  5 end;
  6  /
Trigger created.

tkyte8TKYTE816> create or replace view v
  2  as
  3  select * from TKYTE.tl;
View created.
```

Имеется декларативное требование целостности, явно ссылающееся на таблицу TKYTE.T1, триггер по явно заданной таблице **TKYTE.T4** и представление, явно ссылающееся на **TKYTE.T1.** Давайте экспортируем эту схему и создадим пользователя, в схему которого будет выполняться импорт (учтите, что пользователю, выполняющему импорт с параметрами **FROMUSER** и **TOUSER,** необходимо предоставить роль **IMP_FULL_DATABASE):**

```
tkyte@TKYTE816> host exp userid=tkyte/tkyte owner=tkyte

tkyte@TKYTE816> grant connect, resource to a identified by a;
Grant succeeded.

tkyte@TKYTE816> host imp userid=system/change_on_install fromuser=tkyte
touser=a
```

```
Import: Release 8.1.6.0.0 - Production on Tue Mar 20 09:56:17 2001

(c) Copyright 1999 Oracle Corporation.  All rights reserved.


Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option

JServer Release 8.1.6.0.0 - Production

Export file created by EXPORT:V08.01.06 via conventional path

Warning: the objects were exported by TKYTE, not by you

import done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR character
set
. importing TKYTE's objects into A
. . importing table                                "T1"            0 rows
imported
. . importing table                                "T4"            0 rows
imported
IMP-00041: Warning: object created with compilation warnings
  "CREATE FORCE VIEW "A"."V"                                    ("X") AS "
  "select "X" from TKYTE.t1"
Import terminated successfully with warnings.
```

Теперь уже видно, что с представлением у нас проблема: оно явно ссылается на таблицу TKYTE.T1; пользователь A не может создать представление для этого объекта, поскольку у него нет соответствующих привилегий. К счастью, утилита IMP явно сообщает, что представление создано с ошибкой. Менее понятно, что случилось с декларативным требованием целостности и триггером. Если обратиться к словарю данных, подключившись от имени пользователя A, получим:

```
a@TKYTE816> select table_name, constraint_name,
  2     constraint_type, r_constraint_name
  3     from user_constraints
  4   /

TABLE_NAME      CONSTRAINT_NAME            C  R_CONSTRAINT_NAME

T1              SYS_C002465                P
T4              SYS_C002466                R  SYS_C002465

a@TKYTE816> select trigger_name, table_owner, table_name
  2     from user_triggers
  3   /

TRIGGER_NAME    TABLE_OWNER        TABLE_NAME

T2_TRIGGER      TKYTE              T4
```

Удивительно, но декларативное требование целостности ссылается на таблицу пользователя A. Буква **R** в имени означает ссылку, а требование, на которое она указывает, SYS_C002465, — это первичный ключ таблицы T1 в схеме пользователя A. При наличии в схеме TKYTE уточненного требования целостности ссылок, указывающего на таблицу B.T (таблица T пользователя B), это требование тоже будет импортировано в схе-

му пользователя A как указывающее на таблицу A.T. Если уточненное именем схемы имя таблицы, на которую мы ссылаемся, совпадает с именем владельца в момент экспортирования, утилита EXP это имя не сохранит.

Сравните это с тем, как экспортирован триггер. Триггер T2_TRIGGER создается не по таблице пользователя A — он создается для таблицы пользователя TKYTE! Это потенциально опасный побочный эффект. Если продублировать триггер по таблице TKYTE.T4, его тело будет выполнено дважды, а для таблицы A.T4 триггера вообще не будет.

Настоятельно рекомендую учитывать все это при использовании утилит EXP/IMP для клонирования пользователя. Помните об этих побочных эффектах и ищите их. Выполнив приведенные ниже команды, можно просмотреть все операторы ЯОД, триггеры, процедуры и т.д., прежде чем выполнять их в базе данных:

```
imp userid=sys/manager fromuser=tkyte touser=a INDEXFILE=foo.sql
imp userid=sys/manager fromuser=tkyte touser=a SHOW=Y
```

В крайнем случае попробуйте выполнить импорт в базу данных, где нет учетной записи пользователя FROMUSER. Например, ниже выполняется импорт в базу данных, где имеется пользователь A, но пользователя TKYTE нет:

```
C:\exp>imp userid=sys/manager fromuser=tkyte touser=a

Import: Release 8.1.6.0.0 - Production on Tue Mar 20 10:29:37 2001
(c) Copyright 1999 Oracle Corporation.  All rights reserved.

Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option

JServer Release 8.1.6.0.0 - Production

Export file created by EXPORT:V08.01.06 via conventional path

Warning: the objects were exported by TKYTE, not by you

import done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR character
set
. importing TKYTE's objects into A
. . importing table                "T1"                 0 rows imported
. . importing table                   "T4"              0 rows imported
. . importing table                   "T5"              0 rows imported
IMP-00041: Warning: object created with compilation warnings
  "CREATE FORCE VIEW "A"."V"                                   ("X") AS "
  "select "X" from TKYTE.t1"
IMP-00017: following statement failed with ORACLE error 942:
  "CREATE TRIGGER "A".t2_trigger"
  "before insert on TKYTE.t4"

  "begin"
     null;"
  "end;"
IMP-00003: ORACLE error 942 encountered
ORA-00942: table or view does not exist
Import terminated successfully with warnings.
```

Так можно выяснить наличие подобных побочных эффектов немедленно. Настоятельно рекомендую использовать такой подход для поиска объектов, использующих уточненные схемой имена, и проверки корректности их импортирования.

Несколько иная проблема возникает при импортировании и экспортировании объектных типов Oracle. СУБД Oracle позволяет создавать в базе данных новые типы данных. Можно добавлять типы, равноправные со стандартными NUMBER, DATE, VARCHAR2.... Затем можно создавать таблицы таких типов или таблицы со столбцами таких типов. Поэтому обычный порядок действий — создать схему, затем — типы данных в схеме и, наконец, — объекты, использующие эти типы; причем все это делается от имени одного пользователя. Однако при "клонировании" такой схемы вы столкнетесь с серьезной проблемой. Я продемонстрирую эту проблему, укажу причины ее возникновения и предложу возможное решение.

Начнем со схемы следующего вида:

```
tkyte@TKYTE816> create type my type
  2   as object
  3    (x int,
  4     y date,
  5     z varchar2(20)
  6   )
  7   /
Type created.

tkyte@TKYTE816> create table t1 of my_type
  2   /
Table created.

tkyte@TKYTE816> create table t2  (a int, b my_type);
Table created.

tkyte@TKYTE816> insert into t1 values  (1,  sysdate,  'hello');
1 row created.

tkyte@TKYTE816> insert into t2 values  (55, my_type(1,  sysdate,  'hello'));
1 row created.

tkyte@TKYTE816> commit;
Commit complete.

tkyte@TKYTE816> host exp userid=tkyte/tkyte owner=tkyte
```

Мы получили копию схемы. Попытавшись теперь использовать параметры FROMUSER/TOUSER, можно обнаружить следующее:

```
tkyte@TKYTE816> host imp userid=sys/manager fromuser=tkyte touser=a;

Import: Release 8.1.6.0.0 - Production on Tue Mar 20 12:44:26 2001

(c) Copyright 1999 Oracle Corporation.  All rights reserved.


Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option
JServer Release 8.1.6.0.0 - Production
```

```
Export file created by EXPORT:V08.01.06 via conventional path

Warning: the objects were exported by TKYTE, not by you

import done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR character
set
. importing TKYTE's objects into A
IMP-00017: following statement failed with ORACLE error 2304:
 "CREATE TYPE "MY_TYPE" TIMESTAMP '2001-03-20:12:44:21' OID
"4A301F5AABF04A46"
 "88552E4AF5793176"
 "as object"
 "(x int,"
 " y date,"
 " z varchar2(20) "
 ")"
IMP-00003: ORACLE error 2304 encountered
ORA-02304: invalid object identifier literal
IMP-00063: Warning: Skipping table "A"."T1" because object type
"A"."MY_TYPE" cannot be created or has different identifier
IMP-00063: Warning: Skipping table "A"."T2" because object type
"A"."MY_TYPE" cannot be created or has different identifier
Import terminated successfully with warnings.
```

В этот момент все останавливается. Мы не можем создать тип в пользовательской схеме A, а если бы у нас это и получилось, — это будет **другой** тип, и импортировать данные мы не сможем. В базе данных будут присутствовать как бы два различных типа **NUMBER. В** примере выше мы создаем два разных типа **MY_TYPE,** но работаем с ними, как с одним.

Проблема (к сожалению, в документации она никак не отражена) состоит в том, что не надо создавать схему, содержащую как типы, так и объекты, **особенно** если в дальнейшем планируется экспортировать и импортировать ее подобным образом. Используя принцип создания схем **CTXSYS** и **ORDSYS** для компонента Oracle interMedia, надо создать отдельную схему, содержащую типы. Если необходимо использовать interMedia Text, мы пользуемся типами схемы **CTXSYS.** Если предполагается использовать средства interMedia для работы с изображениями, мы пользуемся типами схемы **ORDSYS.** В нашем случае надо сделать то же самое, т.е. создать схему, содержащую необходимые типы данных:

```
our_types@TKYTE816> connect OUR_TYPES

our_types@TKYTE816> create type my_type
  2  as object
  3  (x int,
  4   y date,
  5   z varchar2(20)
  6  )
  7  /
Type created.

our_types@TKYTE816> grant all on my_type to public;
Grant succeeded.
```

Все схемы должны использовать **эти** типы, а не собственные, которые тесно связаны с их учетной записью. Теперь можно повторно выполнить рассмотренный ранее пример:

```
tkyte@TKYTE816> connect tkyte

tkyte@TKYTE816> create table t1 of our_types.my type
  2  /
Table created.

tkyte@TKYTE816> create table t2 (a int, b our_types.my_type);
Table created.

tkyte@TKYTE816> insert into t1 values (1, sysdate, 'hello');
1 row created.

tkyte@TKYTE816> insert into t2 values (55,
  2  our_types.my_type(1, sysdate, 'hello'));
1 row created.

tkyte@TKYTE816> commit;
Commit complete.

tkyte@TKYTE816> host exp userid=tkyte/tkyte owner=tkyte
```

Итак, единственное отличие в том, что мы используем уточненное имя типа, **OUR_TYPES.MY_TYPE,** а не просто **MY_TYPE.** Поскольку синонимы для типов создавать нельзя, такое уточнение именем схемы обязательно: необходимо указывать полное имя объекта, включая схему, которой он принадлежит. Именно так мы обязаны делать при использовании типов interMedia (например, объектов **CTXSYS** и **ORDSYS),** да и индексов, то есть всегда необходимо указывать уточненные имена. Поэтому тщательно выбирайте имя схемы, содержащей типы, — вам с ним предстоит некоторое время жить!

Теперь давайте рассмотрим, как на этот раз пройдет импортирование:

```
tkyte@TKYTE816> host imp userid=sys/manager fromuser=tkyte touser=a;

Import: Release 8.1.6.0.0 - Production on Tue Mar 20 12:49:33 2001
(c) Copyright 1999 Oracle Corporation.  All rights reserved.

Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option

JServer Release 8.1.6.0.0 - Production

Export file created by EXPORT:V08.01.06 via conventional path

Warning: the objects were exported by TKYTE, not by you
import done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR character set
. importing TKYTE's objects into A
IMP-00017: following statement failed with ORACLE error 2304:
 "CREATE TABLE "T1" OF "OUR_TYPES"."MY_TYPE" OID 'AC60D4D90ED1428B84D245357AD"
 "F2DF3' OIDINDEX (PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE(INITIAL 524288)"
 " TABLESPACE "DATA") PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 LOGGING S"
 "TORAGE(INITIAL 524288) TABLESPACE "DATA""
```

```
IMP-00003: ORACLE error 2304 encountered
ORA-02304: invalid object identifier literal
. . importing table                          "T2"          1 rows
imported
Import terminated successfully with warnings.
```

Уже лучше, но тоже не идеально. Объектную таблицу импортировать не удалось, но реляционная таблица с объектным типом столбца импортирована успешно. Этого следовало ожидать. Объектная таблица теперь формально другая, а для объектов это существенно. Эту проблему можно, однако, обойти, поскольку эти две таблицы фактически созданы по одному и тому же типу. Надо заранее создать объектную таблицу в схеме пользователя A. Можно использовать параметр IMP INDEXFILE= для получения необходимого оператора ЯОД:

```
a@TKYTE816> host imp userid=a/a tables=t1 indexfile=t1.sql
```

Если открыть полученный файл T1.SQL в текстовом редакторе:

```
REM CREATE TABLE "A"."T1" OF "OUR_TYPES"."MY_TYPE" OID
REM '4C60D4D90ED1428B84D245357ADF2DF3' OIDINDEX (PCTFREE 10 INITRANS 2
REM MAXTRANS 255 STORAGE(INITIAL 524288) TABLESPACE "DATA") PCTFREE 10
REM PCTUSED 40 INITRANS 1 MAXTRANS 255 LOGGING STORAGE(INITIAL 524288)
REM TABLESPACE "DATA" ;
REM ALTER TABLE "A"."T1" MODIFY ("SYS_NC_OID$" DEFAULT SYS_OP_GUID()) ;
REM ... 1 rows
```

Необходимо удалить слова REM, а также конструкцию OID xxxxx, а затем выполнить:

```
a@TKYTE816> CREATE TABLE "A"."T1" OF "OUR_TYPES"."MY_TYPE"
  2  OIDINDEX (PCTFREE 10 INITRANS 2
  3  MAXTRANS 255 STORAGE(INITIAL 524288) TABLESPACE "DATA") PCTFREE 10
  4  PCTUSED 40 INITRANS 1 MAXTRANS 255 LOGGING STORAGE(INITIAL 524288)
  5  TABLESPACE "DATA" ;

Table created.

a@TKYTE816> ALTER TABLE "A"."T1" MODIFY ("SYS_NC_OID$" DEFAULT
SYS_OP_GUID()) ;

Table altered.
```

Теперь можно выполнить импорт:

```
a@TKYTE816> host imp userid=a/a tables=t1 ignore=y

Import: Release 8.1.6.0.0 - Production on Tue Mar 20 13:01:24 2001

(c) Copyright 1999 Oracle Corporation.  All rights reserved.


Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option

JServer Release 8.1.6.0.0 - Production

Export file created by EXPORT:V08.01.06 via conventional path

Warning: the objects were exported by TKYTE, not by you
```

```
import done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR character
set
. importing TKYTE's objects into A
. . importing table                                "N1"       1       rows
imported
Import terminated successfully without warnings.
a@TKYTE816> select * from t1;

        X Y          Z

        1 20-MAR-01  hello
```

и данные загружены.

Использование различных версий утилит IMP/EXP

Можно использовать утилиты IMP и EXP с разными версиями СУБД Oracle. Можно даже экспортировать и импортировать данные в СУБД версии 7 и 8. Однако при этом нужно использовать подходящие версии утилит **EXP** и **IMP.** При выборе версии **IMP** и EXP руководствуйтесь следующими правилами:

- Всегда используйте версию **IMP,** соответствующую версии СУБД. Если вы собираетесь импортировать в базу данных версии 8.1.6, это надо делать с помощью утилиты импорта версии 8.1.6.
- Всегда используйте версию **EXP**, соответствующую старешей из двух версий СУБД. Если вы экспортируете данные из версии 8.1.6 в 8.1.5, необходимо использовать версию 8.1.5 утилиты EXP, по протоколу Net8, подключаясь к СУБД версии 8.1.6. Если вы экспортируете из версии 8.1.5 в 8.1.6, необходимо использовать утилиту EXP версии 8.1.5, подключаясь к СУБД версии 8.1.5 локально.

Это принципиально важно. Если попытаться экспортировать из версии 8.1.6 в 8.0.5, например, с помощью утилиты EXP версии 8.1.6, окажется, что утилита **IMP** версии 8.0.5 не может прочитать файл **DMP.** Более того, нельзя использовать версию 8.1.6 утилиты **IMP** для подключения к СУБД версии 8.0.5; это не сработает. В базах данных версии 8.1.6 есть возможности и объекты, не существовавшие в 8.0.5.

Помня о том, что версию IMP определяет СУБД, в которую данные импортируются, а версия EXP должна быть старейшей из двух, всегда можно перенести данные с одной версии на другую.

Последнее замечание: если у вас имеются базы данных Oracle 7, в базах данных Oracle 8 нужно выполнить сценарий, чтобы утилита EXP версии 7 могла с ними работать. Это сценарий **cat7exp.sql**, который находится в каталоге **[ORACLE_HOME]/rdbms/admin.** Сценарий необходимо выполнять от имени пользователя SYS с помощью утилиты командной строки **SVRMGRL. В** результате в базе данных версии 8 будут установлены совместимые с версией 7 сценарии экспорта. Они не заменят представления, используемые для экспорта версией 8, — эти представления останутся нетронутыми.

Сценарий просто добавит в базу данных дополнительные представления версии 7, позволяющие работать утилите EXP версии 7.

Куда делись индексы?

Логично ожидать, что если экспортирована схема, после чего все объекты в этой схеме удалены, а затем схема импортирована, то должен получиться тот же набор объектов. Да, но могут быть сюрпризы. Рассмотрим следующую простую схему:

```
tkyte@TKYTE816> create table t
  2   (x int,
  3    y int,
  4    constraint t_pk primary key(x)
  5   )
  6   /
Table created.

tkyte@TKYTE816> create index t_idx on t(x,y)
  2   /
Index created.

tkyte@TKYTE816> create table t2
  2   (x int primary key,
  3    y int
  4   )
  5   /
Table created.

tkyte@TKYTE816> create index t2_idx on t2(x,y)
  2   /
Index created.
```

Две очень похожие таблицы отличаются наличием явного (а не сгенерированного системой) имени первичного ключа. Давайте посмотрим, какие объекты созданы в базе данных:

```
tkyte@TKYTE816> select object_type, object_name,
  2            decode(status,'INVALID','*','')           status,
  3                    tablespace_name
  4   from user_objects a, user_segments b
  5   where a.object_name = b.segment_name  (+)
  6   order by object_type, object_name
  7   /

OBJECT_TYPE   OBJECT_NAME               S  TABLESPACE_NAME

INDEX         SYS_C002559                  DATA
              T2_IDX                       DATA
              T_IDX                        DATA
              T_PK                         DATA
TABLE         T                            DATA
              T2                           DATA


6 rows selected.
```

Как видите, для каждого из первичных ключей автоматически сгенерирован индекс — это индексы SYS_C002559 и T_PK. Мы видим также и два явно создававшихся индекса. После удаления таблиц T и T2 я выполнил полный импорт. К моему удивлению, обнаружилось следующее:

```
tkyte@TKYTE816> select object_type, object_name,
  2                    decode(status,'INVALID','*',' ')  status,
  3                      tablespace_natne
  4   from user_objects a, user_segments b
  5  where a.object_name = b.segment_name  (+)
  6   order by object_type, object_name
  7  /

OBJECT_TYPE   OBJECT_NAME               S  TABLESPACE_NAME

INDEX         T2_IDX                       DATA
              T_IDX                        DATA
              T_PK                         DATA

TABLE         T                            DATA
              T2                           DATA
```

Один из моих индексов "исчез". Дело в том, что сервер Oracle использовал индекс **T2_IDX** по столбцам (X,Y) для поддержки первичного ключа. Это вполне нормально. Мы и сами можем воспроизвести такое поведение, немного изменив порядок выполнения операторов CREATE (в "чистой" схеме, где еще нет никаких объектов):

```
tkyte@TKYTE816> create table t  (x int, y int) ;
Table created.

tkyte@TKYTE816> create index t_idx on t(x,y);
Index created.

tkyte@TKYTE816> alter table t add constraint t_pk primary key(x);
Table altered.

tkyte@TKYTE816> select object_type, object_name,
  2                   decode(status,'INVALID','*','')  status,
  3                    tablespace_name
  4   from user_objects a, user_segments b
  5  where a.object_name = b.segment_name  (+)
  6  order by object_type, object_name
  7  /

OBJECT_TYPE   OBJECT_NAME               S  TABLESPACE_NAME

INDEX         T_IDX                        DATA
TABLE         T                            DATA
```

В этом случае сервер Oracle будет использовать индекс **T_IDX** для поддержки первичного ключа. Это легко понять, попытавшись удалить индекс:

```
tkyte@TKYTE816> drop index t_idx;
drop index t_idx
```

```
ERROR at line 1:
ORA-02429: cannot drop index used for enforcement of unique/primary key
```

Что ж, то же самое происходит и в ходе работы утилит EXP/IMP. Определения индексов, имена которых сгенерированы системой, не экспортируются. В противном случае происходили бы ошибки. Утилиты **EXP/IMP** полагаются на то, что индекс неявно создается при создании объекта (если вообще создается). Если бы действительно был экспортирован индекс **SYS_C002559,** а затем его попытались бы импортировать, могла бы произойти одна из двух ошибок. Во-первых, сгенерированное имя **SYS_C002559** вполне могло бы совпадать с **уже существующим** автоматически сгенерированным именем (например, именем требования проверки). Во-вторых, при создании объекта индекс мог быть уже создан, то есть наш индекс оказывается лишним (и выдается сообщение об ошибке). Поэтому утилиты EXP и **IMP** работают правильно: вы просто видите побочный эффект того, что при создании требования индекс не **обязательно** создается.

Задавая имя для требования первичного ключа, мы создаем индекс, имя которого совпадает с именем требования; при каждом создании объекта имя индекса будет неизменным. Утилита EXP экспортирует определение этого индекса, а **IMP** будет его импортировать.

Вывод: надо избегать автоматического именования объектов не только по рассмотренной выше причине, но и по причине, описанной в следующем разделе. Не говоря уже о том, что имя **SYS_C002559** никому ни о чем не говорит, а имя **T_PK** для кого-то вполне может означать "первичный ключ (PK) таблицы T".

Явно и автоматически именуемые требования

Еще одна проблема с автоматически сгенерированными именами требований состоит в том, что при импорте для таблицы может добавляться избыточное требование (я мог бы назвать этот раздел "Откуда взялись все эти ограничения?"). Рассмотрим пример. Начнем с таблицы T:

```
tkyte@TKYTE816> create table t
  2  (x int check (x > 5),
  3   y int constraint my_rule check (y > 10),
  4   z int not null ,
  5   a int unique,
  6   b int references t,
  7   c int primary key
  8  );
Table created.

tkyte@TKYTE816> select constraint_name name, constraint_type type,
search_condition
  2     from user_constraints where table_name = 'T';

NAME                           T   SEARCH_CONDITION

SYS_C002S74                    C   "Z" IS NOT NULL
SYS C002675                    C   x > 5
```

```
MY_ROLE                        C  y > 10
SYS_C002677                    P
SYSJC002678                    U
SYS_C002679                    R

6 rows selected.
```

Для нее создано много требований: шесть. Я экспортирую схему, удаляю таблицу и импортирую снова:

```
tkyte@TKYTE816> host exp userid=tkyte/tkyte owner=tkyte

tkyte@tkyte816> drop table T;
Table dropped.

tkyte@TKYTE816> host imp userid=tkyte/tkyte full=y ignore=y rows=n
tkyte@TKYTE816> select constraint_name name, constraint_type type,
search_condition
  2     from user_constraints where table_name =  'T';

NAME                           T   SEARCH_CONDITION

SYS_C002680                    C   "Z"  IS NOT NOLL
SYS_C002681                    C   x > 5
MY_RULE                        C   y > 10
SYS_C002683                    P
SYS_C002684                    U
SYS_C002685                    R

6 rows selected.
```

Пока все выглядит нормально. Предположим, однако, что мы по какой-то причине снова выполняем импорт (например, он частично не удался). При этом мы обнаружим следующее:

```
tkyte@TKYTE816> host imp userid=tkyte/tkyte full=y ignore=y

Import: Release 8.1.6.0.0 - Production on Tue Mar 20 15:42:26 2001

(c) Copyright 1999 Oracle Corporation.  All rights reserved.


Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option
JServer Release 8.1.6.0.0 - Production

Export file created by EXPORT:V08.01.06 via conventional path
import done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR character
set
. importing TKYTE's objects into TKYTE
. . importing table                              "T"           0 rows
imported
IMP-00017: following statement failed with ORACLE error 2264:
 "ALTER TABLE "I" ADD  CONSTRAINT "MY_RULE" CHECK ( y > 10 ) ENABLE
NOVALIDAT"
```

```
IMP-00003: ORACLE error 2264 encountered
ORA-02264: name already used by an existing constraint
IMP-00017: following statement failed with ORACLE error 2261:
  "ALTER TABLE "T" ADD  UNIQUE ("A") USING INDEX PCTFREE 10 INITRANS 2
MAXTRAN"
  "S 255 STORAGE(INITIAL 524288) TABLESPACE "DATA" ENABLE"
IMP-00003: ORACLE error 2261 encountered
ORA-02261: such unique or primary key already exists in the table
About to enable constraints...
Import terminated successfully with warnings.
```

```
tkyte@TKYTE816> select constraint_name name, constraint_type type,
search_condition
  2     from user_constraints where table_name = 'T';

NAME                           T   SEARCH_CONDITION

SYS_C002680                    C   "Z" IS NOT NULL
SYS_C002681                    C   x > 5
MY_RULE                        C   y > 10
SYS_C002683                    P
SYS_C002684                    U
SYS_C002685                    R
SYS_C002686                    C   x > 5

7 rows selected.
```

Появилось дополнительное требование. На самом деле при каждом повторном импортировании будет добавляться дополнительное требование. Для явно поименованного требования было выдано предупреждение: нельзя дважды создавать требование с одним и тем же именем. А вот не имеющее явно заданного имени требование x > 5 создано снова. Так произошло потому, что сервер сгенерировал для него новое имя.

Я знаю случаи, когда выполнялся экспорт на одной базе данных, удалялись данные на другой, и использовался импорт для наполнения структур исходными данными. Со временем накапливались сотни требований проверки для многих столбцов. Производительность начинала падать, и необходимо было выяснить, почему. Причина проста: при каждом копировании данных добавлялся очередной набор требований, делающих те же проверки. Давайте посмотрим, какое влияние может оказать всего сотня избыточных требований:

```
tkyte@TKYTE816> create table t
  2   (x int check (x > 5)
  3   )
  4   /
Table created.

tkyte@TKYTE816> declare
  2       l_start number default dbms_utility.get_time;
  3  begin
  4      for i in 1 .. 1000
  5       loop
  6           insert into t values (10) ;
```

```
  7       end loop;
  8       dbms_output.put_line
  9      (round((dbms_utility.get_time-l_st*rt)/100,2) || ' seconds');
 10  end;
 11  /
.08 seconds

PL/SQL procedure successfully completed.

tkyte@TKYTE816> begin
  2     for i in 1 . . 100
  3       loop
  4            execute immediate
  5           'ALTER TABLE "TKYTE"."T" ADD CHECK ( x > 5 ) ENABLE';
  6       end loop;
  7  end;
  8  /
PL/SQL procedure successfully completed.
tkyte@TKYTE816> declare
  2       l_start number default dbms_utility.get_time;
  3  begin
  4     for i in 1 . . 1000
  5       loop
  6           insert into t values (10) ;
  7       end loop;
  8       dbms_output.put_line
  9       (round((dbms_utility.get_time-l_start)/100,2) || ' seconds');
 10  end;
 11  /
17 seconds

PL/SQL procedure successfully completed.
```

Еще одна веская причина явно задавать имена требований!

Поддержка национальных языков (NLS)

NLS — это сокращение от National Language Support (поддержка национальных языков). Она позволяет хранить, обрабатывать и получать данные на разных языках. Она гарантирует, что утилиты базы данных и сообщения об ошибках, порядок сортировки, форматы представления даты, времени, денежная единица, соглашения по представлению числовых данных и принятый календарь автоматически согласуются с родным языком клиента и форматом. Например, числа выдаются с соответствующим разделителем разрядов и дробной части. В некоторых странах числа должны выдаваться как 999.999.999,99, в других — как 999,999,999.00. Эти особенности необходимо учитывать при использовании утилит экспорта и импорта для переноса данных в среде с различными наборами символов. Существенное значение имеют наборы символов:

- клиента, с которого запускается EXP, и базы данных, из которой данные экспортируются;

- клиента, с которого запускается IMP, и клиента, с которого выполняется EXP;
- клиента, с которого запускается IMP, и базы данных, в которую выполняется импорт.

Если в любой паре наборы символов различаются, данные могут оказаться поврежденными. Рассмотрим весьма тривиальный (но типичный) пример:

```
ops$tkyte@DEV816> create table t  (c varchar2(1));
Table created.

ops$tkyte@DEV816> insert into t values  (chr(235));
1 row created.

ops$tkyte@DEV816> select dump(c)  from t;

DUMP(C)

Typ=1 Len=1: 235

ops$tkyte@DEV816> commit;
Commit complete.
```

Пока все хорошо. Теперь займемся экспортом:

```
ops$tkyte@DEV816> host exp userid=tkyte/tkyte tables=t

Export: Release 8.1.6.2.0 - Production on Tue Mar 20 16:04:55 2001

(c) Copyright 1999 Oracle Corporation.  All rights reserved.


Connected to: Oracle8i Enterprise Edition Release 8.1.6.2.0 - Production
With the Partitioning option
JServer Release 8.1.6.2.0 - Production
Export done in US7ASCII character set and US7ASCII NCHAR character set
server uses WE8ISO8859P1 character set (possible charset conversion)

About to export specified tables via Conventional Path . . .
   . exporting table                              T            1 rows
exported
Export terminated successfully without warnings.
```

Это сообщение (possible charset conversion) должно насторожить! Мы только что взяли 8-битовые данные и экспортировали их в 7-битовый набор символов. Теперь давайте попытаемся импортировать эти данные обратно:

```
ops$tkyte@DEV816> host imp userid=tkyte/tkyte full=y ignore=y

Import: Release 8.1.6.2.0 - Production on Tue Mar 20 16:05:07 2001

(c) Copyright 1999 Oracle Corporation.  All rights reserved.


Connected to: Oracle8i Enterprise Edition Release 8.1.6.2.0 - Production
With the Partitioning option
JServer Release 8.1.6.2.0 - Production
```

```
Export file created by EXPORT:V08.01.06 via conventional path
import done in US7ASCII character set and US7ASCII NCHAR character set
import server uses WE8ISO8859P1 character set (possible charset conversion)
. importing OPS$TKYTE's objects into OPS$TKYTE
. . importing table                            "T"          1 rows
imported

Import terminated successfully without warnings.

ops$tkyte@DEV816> select dump(c)  from t;

DUMP(C)

Typ=1 Len=1: 235
Typ=1 Len=1: 101
```

Функция **DUMP** показывает, что данные, взятые из таблицы и снова в нее помещенные, различаются. Это станет понятнее, если представить числа в двоичном виде:

```
235 в десятичной системе = 11101011 в двоичной
101 в десятичной системе = 01100101 в двоичной
```

Данные были преобразованы из одного набора символов в другой и при этом изменены. Было бы печально обнаружить это **после** удаления таблицы.

Если выдается упомянутое выше сообщение, остановитесь и подумайте о последствиях. В моем случае решение простое. В ОС UNIX или NT достаточно установить значение переменной среды **NLS_LANG** так, чтобы оно соответствовало базе данных:

```
$ echo $NLS_LANG
AMERICAN_AMERICA.WE8ISO8859P1
```

Теперь ни **EXP**, ни **IMP** не будет преобразовывать набор символов. Кроме того, эти утилиты будут заметно быстрее работать. В ОС Windows NT/2000 значение NLS_LANG можно также задать в реестре.

Таблицы, расположенные в нескольких табличных пространствах

Первоначально операторы **CREATE TABLE** были сравнительно простыми. С годами они все более усложнялись. Синтаксическая диаграмма для простого оператора **CREATE TABLE** сейчас занимает восемь страниц. Одно из новейших свойств таблиц — возможность разместить их по частям в нескольких табличных пространствах. Например, таблица со столбцом типа **CLOB** будет иметь сегмент таблицы, сегмент индекса **CLOB** и сегмент данных **CLOB.** Можно явно задать местонахождение таблицы и местонахождение данных столбца типа **CLOB.** Таблица, организованная по индексу, может иметь сегмент индекса и дополнительный сегмент. Фрагментированные таблицы могут состоять из множества фрагментов, каждый из которых располагается в отдельном табличном пространстве.

Это создает определенные трудности для утилит экспорта и импорта. Если попытка импорта объекта завершится неудачно из-за **отсутствия** табличного пространства или ис-

черпания квоты в этом табличном пространстве, утилита IMP автоматически так изменит SQL-операторы, чтобы объект создавался в стандартном табличном пространстве пользователя. Утилита IMP не будет этого делать для объектов, расположенных в нескольких табличных пространствах, даже если все табличные пространства, заданные в команде CREATE, совпадают. Следующий пример это демонстрирует, а затем я опишу, как обойти проблему.

Начнем со схемы, имеющей объекты в нескольких табличных пространствах, и одну простую таблицу, расположенную в одном пространстве:

```
tkyte@TKYTE816> create tablespace exp_test
  2  datafile 'c:\oracle\oradata\tJcyte816\exp_test.dbf'
  3  size 1m
  4  extent management local
  5  uniform size 64k
  6  /
Tablespace created.

tkyte@TKYTE816> alter user tkyte default tablespace exp_test
  2  /
User altered.

tkyte@TKYTE816> create table t1
  2  (x int primary key, y varchar2(25))
  3  organization index
  4  overflow tablespace exp_test
  5  /
Table created.

tkyte@TKYTE816> create table t2
  2  (x int, y clob)
  3  /
Table created.

tkyte@TKYTE816> create table t3
  2 (x int,
  3     a int default to_char(sysdate,'d')
  4  )
  5  PARTITION BY RANGE (a)
  6  (
  7  PARTITION part_1 VALUES LESS THAN(2) ,
  8  PARTITION part_2 VALUES LESS THAN(3),
  9  PARTITION part_3 VALUES LESS THAN(4),
 10  PARTITION part_4 VALUES LESS THAN(5),
 11  PARTITION part_5 VALUES LESS THAN(6),
 12  PARTITION part_6 VALUES LESS THAN(7) ,
 13  PARTITION part_7 VALUES LESS THAN(8)
 14  )
 15  /
Table created.

tkyte@TKYTE816> create table t4 (x int)
  2  /
Table created.
```

Итак, мы создали табличное пространство и установили его в качестве *стандартного* для пользователя. Затем мы создали организованную по индексу таблицу с двумя сегментами — сегментом индекса и дополнительным сегментом. Мы создали таблицу со столбцом типа **CLOB**, состоящую из трех сегментов. Затем мы создали фрагментированную таблицу из семи сегментов. Наконец, имеется еще обычная, "простая" таблица. Выполним экспорт схемы:

```
tkyte@TKYTE816> host exp userid=tkyte/tkyte owner=tkyte
```

и удалим табличное пространство:

```
tkyte@TKYTE816> drop tablespace exp_test including contents;
Tablespace dropped.

tkyte@TKYTE816> alter user tkyte default tablespace data;
User altered.
```

При импорте этой схемы почти все таблицы не будут восстановлены:

```
tkyte@TKYTE816> host imp userid=tkyte/tkyte full=y

Import: Release 8.1.6.0.0 - Production on Tue Mar 20 19:03:18 2001

(c) Copyright 1999 Oracle Corporation.  All rights reserved.


Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option
JServer Release 8.1.6.0.0 - Production

Export file created by EXPORT:V08.01.06 via conventional path
import done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR character sat
. importing TKYTE's objects into TKYTE
IMP-00017: following statement failed with ORACLE error 959:
 "CREATE TABLE "T2" ("X" NUMBER(*,0), "Y" CLOB)  PCTFREE 10 PCTUSED 40 INITRA"
 "NS 1 MAXTRANS 255 LOGGING STORAGE(INITIAL 65536) TABLESPACE "EXP_TEST" LOB "
 " ("Y") STORE AS  (TABLESPACE "EXP_TEST" ENABLE STORAGE IN ROW CHUNK 8192 PCT"
 "VERSION 10 NOCACHE STORAGE(INITIAL 65536))"
IMP-00003: ORACLE error 959 encountered
ORA-00959: tablespace 'EXP_TEST' does not exist
IMP-00017: following statement failed with ORACLE error 959:
 "CREATE TABLE "T3" ("X" NUMBER(*,0), "A" NUMBER(*,0))  PCTFREE 10 PCTUSED 40"
 " INITRANS 1 MAXTRANS 255 LOGGING TABLESPACE "EXP_TEST" PARTITION BY RANGE ("
 ""A" )  (PARTITION "PART_1" VALUES LESS THAN (2) PCTFREE 10 PCTUSED 40 INIT"
 "RANS 1 MAXTRANS 255 STORAGE(INITIAL 65536) TABLESPACE "EXP_TEST" LOGGING, P"
 "ARTITION "PART_2" VALUES LESS THAN (3) PCTFREE 10 PCTUSED 40 INITRANS 1 MA"
 "XTRANS 255 STORAGE(INITIAL 65536) TABLESPACE "EXP_TEST" LOGGING, PARTITION "
 ""PART_3" VALUES LESS THAN (4) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 25"
 "5 STORAGE(INITIAL 65536) TABLESPACE "EXP_TEST" LOGGING, PARTITION "PART_4" "
"VALUES LESS THAN (5)  PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 STORAGE"
 "(INITIAL 65536) TABLESPACE "EXP_TEST" LOGGING, PARTITION "PART_5" VALUES LE"
 "SS THAN (6) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 STORAGE(INITIAL "
```

```
 "65536) TABLESPACE "EXP_TEST" LOGGING, PARTITION "PART_6" VALUES LESS THAN ("
 "7) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 STORAGE(INITIAL 65536) TA"
 "BLESPACE "EXP_TEST" LOGGING, PARTITION "PART_7" VALUES LESS THAN (8) PCTFR"
 "EE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 STORAGE(INITIAL 65536) TABLESPACE "
 ""EXP_TEST" LOGGING )"
IMP-00003: ORACLE error 959 encountered
ORA-00959: tablespace 'EXP_TEST' does not exist
. . importing table "T4" 0 rows imported
IMP-00017: following statement failed with ORACLE error 959:
 "CREATE TABLE "T1" ("X" NUMBER(*,0), "Y" VARCHAR2(25), PRIMARY KEY ("X") EN"
 "ABLE) ORGANIZATION INDEX NOCOMPRESS PCTFREE 10 INITRANS 2 MAXTRANS 255 LOG"
 "GING STORAGE(INITIAL 65536) TABLESPACE "EXP_TEST" PCTTHRESHOLD 50 OVERFLOW"
 "PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 LOGGING STORAGE(INITIAL 6553"
 "6) TABLESPACE "EXP_TEST""
IMP-00003: ORACLE error 959 encountered
ORA-00959: tablespace 'EXP_TEST' does not exist
Import terminated successfully with warnings.
```

Единственная восстановленная безошибочно таблица — это простая "нормальная" таблица. Для этой таблицы утилита IMP переписала SQL-оператор. Она удалила первую обнаруженную конструкцию TABLESPACE EXP_TEST и переписала оператор CREATE. Этот переписанный оператор CREATE успешно выполнился. Другие операторы CREATE, переписанные аналогично, не выполнились. Единственное решение этой проблемы — создать таблицы заранее, а затем импортировать с параметром IGNORE=Y. Если под рукой нет операторов CREATE TABLE для таблиц, их, конечно, можно восстановить из DMP-файла с помощью параметра INDEXFILE=Y. Это позволит изменить операторы, подставив соответствующую информацию о табличных пространствах вручную. В нашем случае, поскольку операторы ЯОД мне известны, я просто создам необходимые три таблицы, указав при необходимости новые табличные пространства:

```
tkyte@TKYTE816> create table t1
  2  (x int primary key, y varchar2(25))
  3  organization index
  4  overflow tablespace data
  5  /
Table created.

tkyte@TKYTE816> create table t2
  2  (x int, y clob)
  3  /
Table created.

tkyte@TKYTE816> create table t3
  2  (x int,
  3   a int default to_char(sysdate,'d')
  4  )
  5  PARTITION BY RANGE (a)
  6  (
  7  PARTITION part_1 VALUES LESS THAN(2),
  8  PARTITION part_2 VALUES LESS THAN(3),
```

```
  9   PARTITION part_3 VALUES LESS THAN(4),
 10   PARTITION part_4 VALUES LESS THAN(5),
 11   PARTITION part_5 VALUES LESS THAN(6),
 12   PARTITION part_6 VALUES LESS THAN(7) ,
 13   PARTITION part_7 VALUES LESS THAN(8)
 14   )
 15   /
Table created.
```

и после этого смогу импортировать данные без ошибок:

```
tkyte@TKYTE816> host imp userid=tkyte/tkyte full=y ignore=y

Import: Release 8.1.6.0.0 - Production on Tue Mar 20 19:03:20 2001

(c) Copyright 1999 Oracle Corporation.  All rights reserved.


Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option
JServer Release 8.1.6.0.0 - Production

Export file created by EXPORT:V08.01.06 via conventional path
import done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR character
set
. importing TKYTE's objects into TKYTE
   . importing table                "T2"                0 rows imported
. . importing partition          "T3":"PART_1"           0 rows imported
. . importing partition          "T3":"PART_2"           0 rows imported
. . importing partition          "T3":"PART_3"           0 rows imported
. . importing partition          "T3":"PART_4"           0 rows imported
   . importing partition          "T3":"PART_5"           0 rows imported
. . importing partition          "T3":"PART_6"           0 rows imported
. . importing partition          "T3":"PART_7"           0 rows imported
   . importing table                   "T4"            0 rows imported
. . importing table         "T1"          0          rows          imported
Import terminated successfully without warnings.
```

Что будет, если для некоторых объектов предложенный обходной путь не подойдет и утилита IMP по-прежнему будет выдавать сообщение об ошибке **ORA-00959 tablespace 'name' does not exist**. Единственное временное решение, которое мне удалось найти, — создать объект заранее (как было показано выше), а затем создать очень небольшие табличные пространства с требуемыми утилитой IMP именами. Для этих табличных пространств надо задать настолько маленькие файлы данных, чтобы в них нельзя было что-то реально создать. Теперь утилита **IMP** будет работать, и в дальнейшем эти табличные пространства можно будет удалить.

Резюме

В этой главе мы рассмотрели много вариантов использования средств экспорта и импорта. Я представил решения типичных проблем и ответы на часто задаваемые вопросы по поводу этих средств. Утилиты **EXP** и **IMP** особенно полезны, если знать пару-тройку

хитрых приемов для работы с ними. С учетом сложности объектов в современных базах данных, я иногда удивляюсь, что они так хорошо работают.

Значение утилит IMP и EXP со временем меняется. Во времена Oracle версий 5 и 6 они считались ценным средством резервного копирования. Базы данных были маленькими (база данных объемом 100 Мбайт считалась большой) а вопрос постоянной доступности данных лишь начал подниматься. Со временем полезность утилит IMP/EXP в сфере резервного копирования и восстановления снизилась до такой степени, что некоторые вообще не считают их средствами резервного копирования и восстановления. Сегодня EXP и IMP — сравнительно простые средства для переноса средних объемов данных с одного экземпляра на другой или (в случае использования переносимых табличных пространств) для множественного переноса данных. Использовать их для резервного копирования базы данных объемом 500 Гбайт — нелепо. А использовать для переноса 100 Гбайт из той же базы данных — вполне допустимо.

По-прежнему применяются они как средство "клонирования" схемы (если вы, конечно, представляете потенциальные проблемы) или для извлечения операторов ЯОД, формирующих схему, в файл. В сочетании с другими возможностями СУБД, например триггерами INSTEAD OF для представлений, с их помощью можно исполнить и ряд новых "трюков". Я уверен, что интересные варианты использования, например совместный доступ двух баз данных к файлам только для чтения, еще ждут своего часа.

9

Загрузка данных

В этой главе мы рассмотрим загрузку данных, другими словами, как поместить данные **в** базу данных Oracle. Глава посвящена, в основном, инструментальному средству **SQL*Loader** (или **SQLLDR,** произносится "сиквел лоудер") как основному методу загрузки данных. Однако по ходу дела будут рассмотрены и другие варианты загрузки, а также кратко описаны способы **извлечения данных** из базы.

Утилита **SQLLDR** была в составе СУБД Oracle, сколько я ее помню, лишь незначительно изменяясь со временем, но при работе с ней у многих все еще возникают вопросы. В этой главе я не собираюсь давать исчерпывающее описание этой утилиты, хочу лишь рассмотреть проблемы, с которыми пользователи сталкиваются ежедневно при ее использовании. Глава построена по принципу "вопрос-ответ". Будет описываться проблема, а затем ее возможные решения. По ходу изложения будут описаны многие особенности использования утилиты **SQLLDR** и загрузки данных вообще:

- загрузка записей с ограничителем и записей в фиксированном формате;
- загрузка дат;
- загрузка данных с использованием последовательностей, а также использование оператора CASE в SQL, добавленного в версии Oracle 8.1.6;
- изменение и загрузка данных в один прием (изменение, если данные существуют, и загрузки — в противном случае);
- загрузка данных с встроенным переводом строк, для которой мы будем использовать ряд новых возможностей и опций (в частности, будут рассмотрены атрибуты **FIX, VAR** и **STR,** добавленные в Oracle 8.1.6);

- загрузка больших объектов с помощью новых типов данных, **BLOB** и **CLOB,** появившихся в Oracle 8.0 и дающих намного больше возможностей, чем старые типы **LONG** и LONG **RAW.**

Мы **не будем** детально описывать режим непосредственной загрузки или использование утилиты **SQLLDR** в среде хранилищ данных, параллельную загрузку и т.д. Эти темы требуют отдельной книги.

Введение в SQL*Loader

Утилита SQL*Loader **(SQLLDR)** — высокопроизводительное средство массовой загрузки данных в СУБД Oracle. Это очень полезное средство, позволяющее поместить в базу данных Oracle данные из текстовых файлов множества различных форматов. Утилиту **SQLLDR** можно использовать для потрясающе быстрой загрузки огромных объемов данных. Она имеет два режима работы:

- **Обычная загрузка. В** этом режиме **SQLLDR** для загрузки данных будет автоматически вставлять строки с помощью SQL-операторов.
- **Непосредственная загрузка. В** этом режиме SQL не используется. Блоки данных в базе формируются непосредственно.

Непосредственная загрузка позволяет читать данные из обычного файла и записывать их непосредственно в сформатированные блоки базы данных в обход SQL-машины (а также сегментов отката и журнала повторного выполнения). При распараллеливании непосредственная загрузка является самым быстрым способом наполнения базы данными, причем ускорить этот процесс невозможно.

Мы не будем детально рассматривать все аспекты использования **SQLLDR.** Описанию утилиты **SQLLDR** посвящено шесть глав руководства *Oracle Server Utilities Guide*. То, что этих глав — шесть, достойно внимания, поскольку каждой из остальных утилит посвящено не более одной главы. Полный синтаксис всех опций можно найти в руководстве, а эта глава посвящена ответам на вопросы типа "как сделать?", не рассмотренным в документации.

Учтите, что интерфейс OCI (Oracle Call Interface для языка C) позволяет создать собственный непосредственный загрузчик на языке C, начиная с первого выпуска версии Oracle 8.1.6 и далее. Эта возможность пригодится, если окажется, что необходимую операцию невозможно выполнить в **SQLLDR** или потребуется интегрировать процесс загрузки в приложение. **SQLLDR** — это утилита командной строки, отдельная программа. Это не библиотека функций, ее нельзя, например, вызвать из PL/SQL.

При вызове **SQLLDR** из командной строки без параметров выдается следующая справочная информация:

```
$ sqlldr

SQLLDR: Release 8.1.6.1.0 - Production on Sun Sep 17 12:02:59 2000
(c) Copyright 1999 Oracle Corporation.  All rights reserved.

Usage: SQLLOAD keyword=value [,keyword=value,...]
```

```
Valid Keywords:

    userid — ORACLE username/password
   control — Control file name
       log — Log file name
       bad — Bad file name
      data — Data file name
   discard — Discard file name
discardmax — Number of discards to allow          (Default all)
      skip — Number of logical records to skip    (Default 0)
      load — Number of logical records to load    (Default all)
    errors — Number of errors to allow            (Default 50)
      rows — Number of rows in conventional path bind array or between
             direct path data saves
              (Default: Conventional path 64, Direct path all)
  bindsize — Size of conventional path bind array in bytes  (Default 65536)
    silent — Suppress messages during run (header, feedback, errors,
             discards, partitions)
    direct — use direct path                            (Default FALSE)
   parfile — parameter file: name of file that contains parameter
             specifications
  parallel — do parallel load                           (Default FALSE)
      file — File to allocate extents from
skip unusable_indexes — disallow/allow unusable indexes or index partitions
                        (Default FALSE)
skip_index_maintenance -- do not maintain indexes, mark affected indexes as
                          unusable   (Default FALSE)
commit_discontinued — commit loaded rows when load is discontinued
                      (Default FALSE)
  readsize — Size of Read buffer                        (Default 1048576)
```

Назначение этих параметров кратко описано в следующей таблице:

Параметр	*Назначение*
BAD	Имя файла, который будет содержать отвергнутые записи по окончании загрузки. Если не указать его имя явно, оно будет создано автоматически по имени управляющего **(CONTROL)** файла (подробнее об управляющих файлах см. далее в этой главе), использованного для загрузки. Например, если в качестве управляющего использован файл **foo.ctl,** файл **BAD** по умолчанию получит имя **foo.bad;** именно в этот файл и будет помещать отвергнутые записи утилита **SQLLDR** (если файл существует, он будет перезаписан).
BINDSIZE	Размер (в байтах) буфера, используемого утилитой **SQLLDR** для вставки данных при обычной загрузке. При непосредственной загрузке этот параметр не используется. Размер буфера используется для определения размера массива, с помощью которого **SQLLDR** будет вставлять данные.

Параметр	*Назначение*
CONTROL	Имя управляющего **(CONTROL)** файла, описывающего формат данных и способ их загрузки в таблицу. Управляющий файл необходимо указывать при каждом вызове **SQLLDR.**
DATA	Имя файла, из которого надо считывать данные.
DIRECT	Допустимы значения True (непосредственная загрузка) и False (обычная загрузка), причем по умолчанию используется False. Таким образом, по умолчанию утилита **SQLLDR** выполняет обычную загрузку.
DISCARD	Имя файла, куда помещаются пропущенные записи, которые не должны загружаться. Утилиту **SQLLDR** можно использовать для фильтрования загружаемых записей — она позволяет загружать только записи, удовлетворяющие указанным критериям.
DISCARDMAX	Задает максимальное количество пропущенных записей, допустимое в процессе загрузки. Если пропущено больше записей, загрузка прекращается. По умолчанию загрузка не прекращается, даже если пропущены все записи.
ERRORS	Максимально допустимое количество ошибок, выявленных утилитой **SQLLDR,** прежде чем загрузка будет прервана. Это могут быть самые разные ошибки, например ошибка преобразования типов данных (скажем, попытка загрузить строку **ABC** в числовое поле), дублирование записей по ключу уникального индекса и т.д. Стандартно допускается не более 50 ошибок, после чего загрузка прекращается. Чтобы можно было в одном сеансе загрузить все допустимые записи (при этом отвергнутые попадают в BAD-файл), укажите в качестве значения большое число, например 999999999.
FILE	При использовании непосредственной загрузки с распараллеливанием, этот параметр позволяет явно указать утилите **SQLLDR,** в какой файл данных загружать записи. Это позволяет уменьшить конфликты доступа к файлам данных при параллельной загрузке и обеспечить запись данных в процессе каждого сеанса загрузки на отдельное устройство.
LOAD	Максимальное количество загружаемых записей. Обычно используется для загрузки небольшого образца данных из большого файла или совместно с параметром **SKIP** для загрузки из входного файла лишь записей определенного диапазона.
LOG	Задает имя журнального **(LOG)** файла. По умолчанию, утилита **SQLLDR** будет создавать журнальный файл с именем, созданным автоматически на основе имени управляющего файла, аналогично **BAD-файлу.**

Параметр	*Назначение*
PARALLEL	Допускаются значения TRUE или FALSE. Если указано значение TRUE, то выполняется параллельная непосредственная загрузка. Этот параметр необязателен при обычной загрузке, ее можно выполнять параллельно и без его установки.
PARFILE	Может использоваться для задания имени файла, содержащего все описываемые параметры в виде пар **КЛЮЧЕВОЕ_СЛОВО=ЗНАЧЕНИЕ.** Это позволяет не задавать параметры в командной строке.
READSIZE	Задает размер буфера, используемого при чтении данных.
ROWS	Количество арок, которое утилита **SQLLDR** должна вставить, прежде чем фиксировать изменения при обычной загрузке. При непосредственной загрузке задает количество строк, которые необходимо загрузить, прежде чем сохранять данные (это аналог фиксации). При обычной загрузке стандартное значение - 64 строки. При непосредственной загрузке по умолчанию данные не сохраняются, пока загрузка не завершена.
SILENT	Подавляет выдачу информационных сообщений в ходе загрузки.
SKIP	Заставляет утилиту **SQLLDR** пропустить указанное в качестве значения этого параметра количество строк во входном файле. Чаще всего используется для продолжения прерванной загрузки (для пропуска уже загруженных записей) или для загрузки только части входного файла.
USERID	Строка подключения к базе данных в формате **ИМЯ_ПОЛЬЗОВАТЕЛЯ/ПАРОЛЬ@БАЗА_ДАННЫХ.** Используется для аутентификации в базе данных.
SKIP_INDEX_MAINTENANCE	Не используется при обычной загрузке, поскольку в этом режиме поддерживаются все индексы. Если этот параметр установлен при непосредственной загрузке, СУБД Oracle не поддерживает индексы: они помечаются как недоступные для использования. После загрузки данных такие индексы необходимо пересоздать.
SKIP_UNUSABLE_INDEXES	Требует от утилиты **SQLLDR** разрешить загрузку строк в таблицу, по которой есть недоступные для использования индексы, если эти индексы - не уникальные.

Чтобы использовать утилиту **SQLLDR,** необходим *управляющий файл.* Управляющий файл содержит информацию, описывающую загружаемые данные: их организацию, типы денных и т.д., а также указывает, в какую таблицу или таблицы эти данные необходимо

загрузить. Управляющий файл может содержать даже данные, которые необходимо загрузить. В следующем примере создается простой управляющий файл и описываются используемые на каждом шаге команды:

```
LOAD DATA
```

LOAD DATA. Эта команда указывает утилите **SQLLDR,** что необходимо сделать (в данном случае — загрузить данные). А еще можно указывать действие **CONTINUE_LOAD** для возобновления загрузки. Эта опция используется только для продолжения непосредственной загрузки нескольких таблиц.

```
INFILE *
```

INFILE *. Эта конструкция указывает **SQLLDR,** что данные, которые необходимо загрузить, находятся в самом управляющем файле (см. ниже). В этой конструкции можно также указать имя другого файла, содержащего данные. При необходимости в командной строке можно переопределить имя файла, задаваемого в конструкции **INFILE.** Учтите, что **опции командной строки имеют преимущество над установками, заданными в управляющем файле,** как будет рассмотрено в разделе "Предупреждения".

```
INTO TABLE DEPT
```

INTO TABLE DEPT. Эта конструкция указывает, в какую таблицу загружаются данные; в нашем случае это таблица **DEPT.**

```
FIELDS TERMINATED BY ','
```

FIELDS TERMINATED BY ','. Эта конструкция указывает, что данные будут представлены в виде списка значений через запятую. Есть десятки способов описать загружаемые данные в **SQLLDR,** это — лишь один из наиболее часто используемых.

```
(DEPTNO,
   DNAME,
   LOC
)
```

(DEPTNO, DNAME, LOC). Эта конструкция указывает, какие столбцы загружаются, их порядок следования в загружаемых данных и типы. При этом указываются типы данных во входном потоке, а не типы соответствующих столбцов в базе данных. В нашем случае используется стандартный формат **CHAR(255),** что вполне подходит.

```
BEGINDATA
```

BEGINDATA. Эта конструкция указывает утилите **SQLLDR,** что описание загружаемых данных закончено и что со следующей строки идут данные, которые необходимо загрузить в таблицу **DEPT:**

```
10,Sales,Virginia
20.Accounting,Virginia
30,Consulting,Virginia
40,Finance,Virginia
```

Итак, вот управляющий файл в одном из наиболее простых и типичных форматов — для загрузки в таблицу данных со столбцами, определяемыми разделителем. В этой гла-

ве будут рассмотрены намного более сложные примеры, но для начального знакомства он вполне пригоден. Чтобы применить этот управляющий файл, достаточно создать пустую таблицу **DEPT:**

```
tkyte@TKYTE816> create table dept
  2   (deptno  number(2)  constraint emp_pk primary key,
  3    dname    varchar2(14),
  4    loc          varchar2(13)
  5  )
  6  /
```

Table created.

и выполнить следующую команду:

```
C:\sglldr>sqlldr userid=tkyte/tkyte control=demo1.ctl

SQLLDR: Release 8.1.6.0.0 - Production on Sat Apr 14 10:54:56 2001
(c) Copyright 1999 Oracle Corporation.  All rights reserved.
Commit point reached - logical record count 4
```

Если таблица не пуста, будет выдано сообщение об ошибке:

```
SQLLDR-601: For INSERT option, table must be empty.  Error on table DEPT
```

Так происходит потому, что в управляющем файле используются почти исключительно стандартные установки, а стандартно при загрузке выполняется операция **INSERT** (еще возможны APPEND, TRUNCATE или REPLACE). При выполнении операции INSERT предполагается, что таблица пуста. Если необходимо **добавить** записи в таблицу **DEPT,** можно указать операцию **APPEND,** а для замены данных в таблице **DEPT** — операцию **REPLACE** или TRUNCATE.

При каждой попытке загрузки генерируется журнальный файл. Журнальный файл для нашего примера будет иметь следующий вид:

```
SQLLDR: Release 8.1.6.0.0 - Production on Sat Apr 14 10:58:02 2001

(c) Copyright 1999 Oracle Corporation.  All rights reserved.

Control File:   demo1.ctl

Data File:      demo1.ctl
  Bad File:     demo1.bad
  Discard File: none specified

 (Allow all discards)

Number to load: ALL
Number to skip: 0
Errors allowed: 50
Bind array:     64 rows, maximum of 65536 bytes
Continuation:   none specified
Path used:      Conventional

Table DEPT, loaded from every logical record.
Insert option in effect for this table: INSERT
```

```
    Column Name      Position   Len Term Encl  Datatype

DEPTNO                  FIRST     *   ,        CHARACTER
DNAME                    NEXT     *   ,        CHARACTER
LOC                      NEXT     *   ,        CHARACTER

Table DEFT:
   4 Rows successfully loaded.
   0 Rows not loaded due to data errors.
   0 Rows not loaded because all WHEN clauses were failed.
   0 Rows not loaded because all fields were null.


Space allocated for bind array:              49536 bytes(64 rows)
Space allocated for memory besides bind array:          0 bytes

Total logical records skipped:         0
Total logical records read:            4
Total logical records rejected:        0
Total logical records discarded:       0

Run began on Sat Apr 14 10:58:02 2001
Run ended on Sat Apr 14 10:58:02 2001

Elapsed time was:          00:00:00.11
CPU time was:              00:00:00.04
```

В журнальном файле представлены различные характеристики выполненной загрузки. Можно увидеть использованные опции (стандартные или явно указанные). Можно узнать, сколько записей прочитано, сколько из них было загружено и т.д. Указаны имена файлов **BAD** и **DISCARD.** Указана даже продолжительность загрузки. Журнальные файлы позволяют проверить, успешно ли прошла загрузка, и не было ли сообщений об ошибках. Если при загрузке данных были ошибки SQL (загружаемые данные — "плохие", и соответствующие записи помещены в BAD-файл), эти ошибки записываются в журнальный файл. Информация в журнальном файле не нуждается в комментариях, поэтому мы не будем его рассматривать.

Как сделать...

Теперь мы переходим к наиболее часто задаваемым, по моему опыту, вопросам о загрузке и выгрузке данных из базы данных Oracle с помощью утилиты **SQLLDR.**

Загрузка данных с разделителями

Данные с разделителями, т.е. разделенные некоторым специальным символом и, возможно, взятые в кавычки, — наиболее популярный в настоящее время формат обычных файлов. На больших ЭВМ чаще всего используются файлы с записями фиксированной длины и фиксированного формата, но в UNIX и NT нормой при обмене данными является текстовый файл с разделителями. В этом разделе мы изучим популярные опции, используемые при загрузке данных с разделителями.

Наиболее популярным форматом данных с разделителями является формат CSV (comma-separated values — значения через запятую). При этом формате файла, когда каждое поле данных отделяется от следующего запятой, текстовые строки можно брать в кавычки, так что в самой строке данных может содержаться запятая. Если в строке также должен быть символ кавычки, принято соглашение удваивать такой символ кавычки (в представленном ниже коде мы использовали "", а не просто ").

Типичный управляющий файл для загрузки данных с разделителями выглядит примерно так:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
(DEPTNO,
DNAME,
LOC
)
BEGINDATA
10,Sales,"""USA"""
20,Accounting,"Virginia,USA"
30,Consulting,Virginia
40,Finance,Virginia
50,"Finance","",Virginia
60,"Finance",,Virginia
```

Основное содержится в следующей строке:

```
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY ""
```

Она указывает, что поля данных разделяются запятыми, и что каждое поле **может** быть взято в двойные кавычки. При запуске утилиты **SQLLDR** с этим управляющим файлом будут получены следующие результаты:

```
tkyte@TKYTE816> select * from dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 Sales          "USA"
        20 Accounting     Virginia,USA
        30 Consulting     Virginia
        40 Finance        Virginia
        50 Finance
        60 Finance

6 rows selected.
```

Обратите внимание на следующее:

- **"USA"** — это результат, полученный из загружаемых данных вида """USA""". Утилита **SQLLDR** восприняла два вхождения " как одну кавычку внутри строки в кавычках. Для загрузки данных с необязательными символами, используемыми вокруг строк, необходимо эти символы удваивать.

- **Virginia,USA** в качестве местонахождения отдела 20 — это результат, полученный при загрузке данных вида **"Virginia,USA".** Это поле загружаемых данных пришлось взять в кавычки, чтобы можно было загрузить запятую как часть данных. Иначе запятая считалась бы признаком конца поля, и слово **Virginia** было бы загружено без уточнения USA.
- Записи для отделов 50 и 60 были загружены с пустыми (Null) значениями в поле местонахождения. Если данных нет, можно указывать пустую строку в кавычках, а можно и не указывать — результат будет тем же.

Еще один популярный формат — данные, разделяемые символами табуляции. Есть два способа загрузки таких данных с помощью конструкции **TERMINATED BY:**

- **TERMINATED BY X'09',** т.е. символ табуляции задается в шестнадцатиричном виде (9 — это ASCII-код символа табуляции), но можно также использовать конструкцию
- TERMINATED BY **WHITESPACE**

Интерпретация этих двух вариантов, однако, существенно различается, как будет продемонстрировано ниже. Используя представленную выше таблицу **DEPT,** мы попытаемся загрузить данные с помощью следующего управляющего файла:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY WHITESPACE
 (DEPTNO,
DNAME,
LOC)
BEGINDATA
10       Sales       Virginia
```

На странице это может быть неочевидным, но между каждым приведенным выше фрагментом данных указаны **два** символа табуляции. Строка данных фактически имеет вид:

```
10\t\tSales\t\tVirginia
```

где \t — стандартная управляющая последовательность для представления символа табуляции. При использовании управляющего файла с конструкцией **TERMINATED BY WHITESPACE,** как показано выше, в таблице **DEPT** окажутся следующие данные:

```
tkyte@TKYTE816>  select  *  from dept;

    DEPTNO  DNAME        LOC
----------  -----------  -------------
        10  Sales        Virginia
```

Если указана конструкция **TERMINATED BY WHITESPACE,** в строке ищется первое вхождение пробельного символа (табуляции, пробела или перевода строки), после чего разделителем считается все вплоть до следующего непробельного символа. Поэто-

му при разборе данных **DEPTNO** получает значение 10, а два следующих символа табуляции считаются разделителем, затем значение **Sales** присваивается столбцу **DNAME** и так далее.

Если указать конструкцию **FIELDS TERMINATED BY X'09'** как в следующем управляющем файле:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY X'09'
(DEPTNO,
DNAME,
LOC
)
BEGINDATA
10       Sales       Virginia
```

то окажется, что в таблицу **DEPT** загружены следующие данные:

```
tkyte@TKYTE816> select * from dept;

    DEPTNO DNAME       LOC
---------- ----------- -------------
        10             Sales
```

В данном случае, как только утилита **SQLLDR** обнаруживает символ табуляции, она выдает значение. Поэтому значение 10 присвоено столбцу **DEPTNO,** а в столбце **DNAME** оказалось значение Null, поскольку после первого символа табуляции никаких данных не было, сразу стоял следующий символ табуляции. Значение **Sales** попало в столбец LOC.

Так и должны работать конструкции **TERMINATED BYWHITESPACE** и **TERMINATED BY <символ>**. Загружаемые данные и способ их интерпретации определяют, какую из них использовать.

При загрузке данных с разделителями такого вида часто необходимо пропустить некоторые столбцы загружаемых записей. Например, может потребоваться загрузить столбцы 1, 3 и 5, пропустив столбцы 2 и 4. Для этого утилита **SQLLDR** предлагает ключевое слово **FILLER.** Оно позволяет сопоставить столбец полю входной записи, но не помещать его в базу данных. Например, можно загрузить в представленную ранее таблицу **DEPT** данные, содержащие четыре поля, с использованием следующего управляющего файла, не помещающего второе поле в базу данных:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ' , ' OPTIONALLY ENCLOSED BY ""
( DEPTNO,
  FILLER_1 FILLER,
  DNAME,
```

```
   LOC
)
BEGINDATA
20,Something Not To Be Loaded,Accounting,"Virginia,USA"
```

В результате получится следующая таблица DEPT:

```
tkyte@TKYTE816> select * from dept;

    DEPTNO DNAME          LOC

        20 Accounting     Virginia,USA
```

Загрузка данных в фиксированном формате

Часто имеется текстовый файл, который сгенерирован внешней системой и содержит записи фиксированной длины с данными в фиксированных символьных позициях. Например, поле NAME задано в позициях с 1 по 10, поле ADDRESS — позициях с 11 по 35 и т.д. Давайте рассмотрим, как с помощью утилиты SQLLDR импортировать такие данные.

Эти данные в виде записей фиксированной длины с полями в заранее известных позициях являются оптимальными для загрузки с помощью SQLLDR. Они обрабатываются быстрее всего, поскольку поток данных очень просто анализировать. Утилита SQLLDR будет читать данные с фиксированных позиций фиксированной длины и помещать их в поля, что сделать очень просто. Если необходимо загрузить очень большой объем данных, лучше всего преобразовать их в формат с фиксированными полями. Недостатком файла в фиксированном формате является, конечно же, то, что он может быть намного больше по размеру, чем соответствующий файл в формате с разделителями.

Для загрузки данных с фиксированных позиций, необходимо использовать в управляющем файле ключевое слово POSITION. Например:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
(DEPTNO  position(1:2),
 DNAME   position(3:16),
 LOC     position(17:29)
)
BEGINDATA
10Accounting      Virginia,USA
```

В этом управляющем файле не используется конструкция FIELDS TERMINATED BY, вместо этого для указания начала и конца полей используется конструкция POSITION. Следует отметить, что с помощью конструкции POSITION можно задавать перекрывающиеся поля, в любом порядке. Например, если бы пришлось изменить таблицу DEPT следующим образом:

```
tkyte@TKYTE816> alter table dept add entire_line varchar(29);
Table altered.
```

то затем для загрузки можно было бы использовать управляющий файл:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
(DEPTNO      position(1:2),
 DNAME       position(3:16) ,
 LOC         position(17:29),
 ENTIRE_LINE position(1:29)
)
BEGINDATA
10Accounting     Virginia,USA
```

Позиции поля **ENTIRE_LINE** определяются как (1:29) — в него попадают все 29 байт данных, тогда как другие поля — всего лишь подстроки загружаемых данных. Результат применения представленного выше управляющего файла будет таким:

```
tkyte@TKYTE816> select * from dept;

   DEPTNO DNAME       LOC            ENTIRE_LINE

       10 Accounting  Virginia,USA    10Accounting    Virginia,USA
```

При использовании конструкции POSITION, можно задавать как абсолютные, так и относительные смещения. В представленном выше примере использовались абсолютные смещения. Я явно указывал, где начинаются поля и где они заканчиваются. Этот управляющий файл можно переписать следующим образом:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
(DEPTNO      position(1:2) ,
 DNAME       position(*:16),
 LOC         position(*:29),
 ENTIRE_LINE position(1:29)
)
BEGINDATA
10Accounting     Virginia,USA
```

Символ * требует начать поле с той позиции, перед которой закончилось предыдущее. Поэтому в нашем случае (*:16) — это то же самое, что и (3:16). Учтите, что можно задавать как абсолютные, так и относительные позиции в одном управляющем файле. Кроме того, при использовании символа * можно добавлять к смещению константы. Например, если поле DNAME начинается через два символа после завершения поля DEPTNO, можно использовать конструкцию (*+2:16). В нашем примере это будет идентично использованию конструкции (5:16).

Завершающая позиция поля в конструкции POSITION должна задаваться как абсолютная, с начала записи. Иногда может быть проще указать начальную позицию и длину поля, особенно если поля идут подряд, как в предыдущем примере. При этом доста-

точно указать утилите **SQLLDR,** что запись начинается с символа 1, а затем задать длину каждого поля. Это позволит не вычислять смещения начала и конца каждого поля от начала записи, что иногда затруднительно. Для этого завершающая позиция поля не указывается, а вместо нее задается **длина** каждого поля в записи фиксированного формата, как показано ниже:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
( DEPTNO       position(1)  char(2),
  DNAME        position(*)  char(14),
  LOC          position(*)  char(13) ,
  ENTIRE_LINE  position(1)  char(29)
)
BEGINDATA
10Accounting      Virginia,USA
```

Здесь пришлось указать утилите **SQLLDR,** где начинается первое поле и его длину. Каждое последующее поле начинается со следующей позиции после завершения предыдущего и имеет указанную длину. Абсолютные позиции надо указать только для последнего поля, поскольку оно начинается с начала записи.

Загрузка дат

Загрузка дат с помощью **SQLLDR** выполняется достаточно просто, но у многих вызывает вопросы. Нужно просто указать тип данных **DATE** в управляющем файле и соответствующую маску формата даты. Маска формата даты такая же, как и для функций **TO_CHAR** и **TO_DATE** в SQL Утилита **SQLLDR** применит эту маску к данным и загрузит их автоматически.

Например, если еще раз изменить нашу таблицу **DEPT:**

```
tkyte@TKYTE816> alter table dept add last_updated date;

Table altered,
```

то можно будет загрузить в нее данные с помощью следующего управляющего файла:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY  ','
(DEPTNO,
  DNAME,
  LOC,
  LAST_UPDATED  date  'dd/mm/yyyy'
)
BEGINDATA
10,Sales,Virginia,1/5/2000
20,Accounting,Virginia,21/6/1999
```

```
30,Consulting,Virginia,5/1/2000
40,Finance,Virginia,15/3/2001
```

В результате, таблица **DEPT** будет выглядеть так:

```
tkyte@TKYTE816> select * from dept;
    DEPTNO  DNAME        LOC          LAST_UPDA
-----------------------------------------------
        10  Sales        Virginia     01-MAY-00
        20  Accounting   Virginia     21-JON-99
        30  Consulting   Virginia     05-JAN-00
        40  Finance      Virginia     15-MAR-01
```

Вот так просто. Достаточно указать формат даты в управляющем файле и утилита **SQLLDR** автоматически преобразует данные в даты. В некоторых случаях может потребоваться использовать более сложные функции SQL. Например, если входной файл содержит даты в различных форматах: иногда с компонентом времени, иногда — без, иногда в формате **DD-MON-YYYY,** иногда — в формате **DD/MM/YYYY** и т.д. **В** следующем разделе мы рассмотрим, как с помощью функций в **SQLLDR** решить эти проблемы.

Загрузка данных с использованием последовательностей и других функций

В этом разделе мы рассмотрим, как использовать последовательности и функции SQL при загрузке данных. Помните, однако, что для использования последовательностей и функций SQL необходимо обращаться к SQL-машине, поэтому при непосредственной загрузке использовать их нельзя.

Использовать функции в **SQLLDR** очень просто, если понимать, как утилита **SQLLDR** строит свои операторы **INSERT.** Чтобы применить функцию к полю в **SQLLDR,** необходимо просто добавить ее в управляющем файле в двойных кавычках. Предположим, необходимо обеспечить загрузку данных в верхнем регистре в рассмотренную ранее таблицу **DEPT.** Для этого можно использовать следующий управляющий файл:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ' , '
 (DEPTNO,
   DNAME           "upper(:dname)",
   LOC              "upper(:loc)",
   LAST_UPDATED date   'dd/mm/yyyy'
)
BEGINDATA
10,Sales,Virginia,1/5/2000
20,Accounting,Virginia,21/6/1999
30,Consulting,Virginia,5/1/2000
40,Finance,Virginia,15/3/2001
```

В результате, в базе данных окажется следующее:

```
tkyte@TKYTE816> select * from dept;

 DEPTNO  DNAME          LOC        ENTIRE_LINE        LAST_UPDA

     10  SALES          VIRGINIA                      01-MAY-00
     20  ACCOUNTING     VIRGINIA                      21-JUN-99
     30  CONSULTING     VIRGINIA                      05-JAN-00
     40  FINANCE        VIRGINIA                      15-MAR-01
```

Обратите внимание, как просто перевести данные в верхний регистр — достаточно применить функцию **UPPER** к связываемой переменной. Учтите, что в SQL-функциях можно использовать любые столбцы, независимо от того, значение какого столбца определяет функция. Это означает, что значение столбца может задаваться как функция от нескольких других столбцов. Например, если надо загрузить данные в столбец **ENTIRE_LINE,** можно было бы использовать оператор конкатенации SQL. В данном случае все несколько сложнее, чем кажется. Пока что в записях загружаемых данных — четыре поля. Если просто добавить **ENTIRE_LINE** в управляющий файл следующим образом:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ' , '
(DEPTNO,
  DUMB           "upper(:dname)",
  LOC             "upper(:loc)",
  LAST_UPDATED date 'dd/mm/yyyy',
  ENTIRE_LINE    ":deptno||:dname||:loc||:last_updated"
)
BEGINDATA
10,Sales,Virginia,1/5/2000
20,Accounting,Virginia,21/6/1999
30,Consulting,Virginia,5/1/2000
40,Finance,Virginia,15/3/2001
```

то для каждой загружаемой записи в журнальном файле будет зарегистрирована следующая ошибка:

```
Record 1: Rejected - Error on table DEPT, column ENTIRE_LINE.
Column not found before end of logical record (use TRAILING NULLCOLS)
```

В данном случае утилита **SQLLDR** сообщает, что данных в записи (полей) не хватило для всех столбцов. Решение этой проблемы — простое, и утилита **SQLLDR** даже подсказывает, что делать: USE TRAILING NULLCOLS. В результате SQLLDR будет связывать значение Null со столбцом данных, если в загружаемой записи данных для него нет. В данном случае добавление **TRAILING NULLCOLS** приведет к тому, что связываемая переменная :ENTIRE_LINE будет получать значение Null. Поэтому мы попробуем еще раз, со следующим управляющим файлом:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ','
TRAILING NULLCOLS
(DEPTNO,
  DNAME          "upper(:dname)",
  LOC             "upper(:loc)",
  LAST_UPDATED date 'dd/mm/yyyy',
  ENTIRE_LINE   ":deptno||:dname||:loc||:last_updated"
)
BEGINDATA
10,Sales,Virginia,1/5/2000
20,Accounting,Virginia,21/6/1999
30,Consulting,Virginia,5/1/2000
40,Finance,Virginia,15/3/2001
```

Теперь в таблице будут такие данные:

```
tkyte@TKYTE816> select * from dept;

 DEPTNO DNAME        LOC         ENTIRE_LINE                      LAST_UPDA

     10 SALES        VIRGINIA    10SalesVirginial/5/2000          01-MAY-00
     20 ACCOUNTING   VIRGINIA    20AccountingVirginia21/6/1999    21-JUN-99
     30 CONSULTING   VIRGINIA    30ConsultingVirginia5/1/2000     05-JAN-OO
     40 FINANCE      VIRGINIA    40FinanceVirginial5/3/2001       15-MAR-01
```

Понять, почему этот трюк удался, можно, если разобраться, как утилита **SQLLDR** строит операторы **INSERT.** Утилита **SQLLDR** просматривает заданный управляющий файл и находит в нем столбцы **DEPTNO, DNAME, LOC, LAST_UPDATED** и **ENTIRE_LINE.** Затем она создает пять связываемых переменных, имена которых совпадают с именами столбцов. Обычно, если функции не используются, она строит следующий простой оператор INSERT:

```
INSERT INTO DEPT (DEPTNO, DNAME, LOC, LAST_UPDATED, ENTIRE_LINE)
VALUES (:DEPTNO, :DNAME, :LOC, :LAST_UPDATED, :ENTIRE_LINE) ;
```

Затем она разбирает входной поток, присваивает значения связываемым переменным и выполняет оператор. При использовании функций, **SQLLDR** включает их в оператор **INSERT. В** рассмотренном выше примере оператор **INSERT,** созданный утилитой **SQLLDR,** будет выглядеть так:

```
INSERT INTO T (DEPTNO, DNAME, LOC, LAST_UPDATED, ENTIRE_LINE)
VALUES (:DEPTNO, upper(:dname) , upper(:loc), :last_updated,
           :deptno||:dname||:loc||:last_updated);
```

Поэтому практически все, что можно придумать и выполнить в операторе SQL, можно включить и в управляющий файл **SQLLDR.** С учетом наличия оператора **CASE** в SQL (он добавлен в Oracle 8i), можно создавать мощные и при этом простые схемы загрузки. Предположим, необходимо загрузить даты, которые иногда содержат компонент времени, а иногда — нет. Для этого можно использовать управляющий файл следующего вида:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ','
TRAILING NULLCOLS
(DEPTNO,
  DNAME          "upper(:dname)",
  LOC             "upper(:loc)",
  LAST_UPDATED "case when length(:last_updated) <= 10
                        then to_date(:last_updated,'dd/mm/yyyy')
                        else to_date(:last_updated,'dd/mm/yyyy hh24:mi:ss')
                   end"
)
BEGINDATA
10,Sales,Virginia,1/5/2000 12:03:03
20,Acoounting,Virginia,21/6/1999
30,Consulting,Virginia,5/1/2000 01:23:00
40,Finance,Virginia,15/3/2001
```

что дает в результате:

```
tkyte@TKYTE816> alter session
  2                  set nls_date_format =  'dd-mon-yyyy hh24:mi:ss';
Session altered.
tkyte@TKYTE816> select * from dept;

 DEPTNO DNAME          LOC          ENTIRE_LINE    LAST_UPDATED

     10 SALES          VIRGINIA                    01-may-2000 12:03:03
     20 ACCOUNTING     VIRGINIA                    21-jun-1999 00:00:00
     30 CONSULTING     VIRGINIA                    05-jan-2000 01:23:00
     40 FINANCE        VIRGINIA                    15-mar-2001 00:00:00
```

Теперь к строкам загружаемых данных будет применяться один из двух форматов даты (обратите внимание, что теперь мы загружаем **не даты,** а обычные строки). Оператор CASE выбирает одну из масок формата даты в зависимости от длины строки.

Можно создавать **собственные** функции для вызова при загрузке с помощью **SQLLDR.** Это прямое следствие того факта, что PL/SQL-функции можно вызывать в SQL. Допустим, даты во входном файле указаны в одном из следующих форматов (удивительно, как часто такое бывает — почему-то принято в файлах данных использовать несколько форматов дат):

```
dd-mon-yyyy
dd-month-yyyy
dd/mm/yyyy
dd/mm/yyyy hh24:mi:ss
```

количество секунд, прошедшее после 1 января 1970 года по Гринвичу (или "UNIX-время")

Теперь использовать оператор **CASE** очень сложно, поскольку по длине строки нельзя однозначно определить ее формат. Вместо этого можно создать функцию, которая бу-

дет перебирать форматы даты, пока не найдет подходящий. Следующая функция в цикле перебирает возможные форматы даты, применяя их поочередно, пока не удастся успешно выполнить преобразование. Если по завершении цикла преобразовать строку в дату не удалось, мы предполагаем, что это дата в формате OC UNIX, и выполняем соответствующее преобразование. Если и это не получится, информация об ошибке просто передается утилите SQLLDR, которая поместит соответствующую запись в BAD-файл. Функция выглядит следующим образом:

```
tkyte@TKYTE816> create or replace
  2   function my_to_date(p_string in varchar2)  return date
  3   as
  4        type fmtArray is table of varchar2(25);
  5
  6        l_fmts   fmtArray  :=  fmtArray('dd-mon-yyyy',  'dd-month-yyyy',
  7                                                  'dd/nm/yyyy',
  8                        'dd/mm/yyyy                          hh24 :mi: ss');
  9        l_return date;
 10   begin
 11       for i in 1  ..  l_fmts.count
 12        loop
 13             begin
 14                  l_return  :=  to_date(p_string,  l_fmts(i));
 15             exception
 16                  when  others  then null;
 17             end;
 18             EXIT when l_return is not null;
 19        end loop;
 20
 21        if  (l_return is null)
 22        then
 23             l_return  :=
 24          new_time(to_date('01011970','ddmmyyyy')          +  1/24/60/60  *
 25                              p_string,  'GMT',  'EST');
 26        end if;
 27
 28        return l_return;
 29   end;
 30   /

Function created.
```

Теперь ее можно использовать в управляющем файле:

```
LOAD DATA
INFILE *
INTO TABLE DEFT
REPLACE
FIELDS TERMINATED BY ' , '
(DEPTNO,
  DNAME         "upper(:dname)",
  LOC           "upper(:loc)",
  LAST_UPDATED "my_to_date(:last_updated) "
```

```
BEGINDATA
10,Sales,Virginia,01-april-2001
20,Accounting,Virginia,13/04/2001
30,Consulting,Virginia,14/04/2001 12:02:02
40,Finance,Virginia,987268297
50,Finance,Virginia,02-apr-2001
60,Finance,Virginia,Not a date
```

После загрузки в журнальном файле окажется следующее сообщение об ошибке:

```
Record 6: Rejected - Error on table DEPT, column LAST_UPDATED.
ORA-06502: PL/SQL: numeric or value error: character to number conversion
   error
ORA-06512: at "TKYTE.MY_TO_DATE", line 30
ORA-06512: at line 1
```

показывающее, что последнюю запись загрузить не удалось, но все остальные были загружены. Эта не загруженная запись окажется в BAD-файле. Ее можно исправить и загрузить повторно. Если проверить загруженные данные, получим:

```
tkyte@TKYTE816> alter session
  2                  set nls_date_format = 'dd-mon-yyyy hh24:mi:ss';

Session altered.

tkyte@TKYTE816> select deptno, dname, loc, last_updated from dept;

 DEPTNO      DNAME        LOC          LAST UPDATED
-----------------------------------------------------------
     10  SALES           VIRGINIA     01-apr-2001 00:00:00
     20  ACCOUNTING      VIRGINIA     13-apr-2001 00:00:00
     30  CONSULTING      VIRGINIA     14-apr-2001 12:02:02
     40  FINANCE         VIRGINIA     14-apr-2001 12:11:37
     50  FINANCE         VIRGINIA     02-apr-2001 00:00:00
```

Изменение существующих строк и вставка новых

Часто приходится добавлять записи из файла в таблицу. При этом данные используются для обновления существующих строк, задаваемых первичным ключом, или просто вставить их, если таких строк еще нет. За один шаг это сделать нельзя, но вполне можно за три, причем простых. Сначала я кратко опишу, что нужно сделать, а затем шаг за шагом продемонстрирую соответствующий код. Итак, необходимо.

1. Загрузить все данные с опцией APPEND, указав параметр **ERRORS=99999999**. Задание большого количества возможных ошибок позволит загрузить все "хорошие" (новые) записи. Записи, которые являются обновлениями существующих, будут отвергнуты из-за нарушения ограничения первичного ключа. Они будут записаны в BAD-файл. Итак, все новые записи уже загружены.

2. Загрузить BAD-файл в рабочую таблицу с опцией TRUNCATE. Структура этой таблицы совпадает со структурой "реальной" — она должна иметь тот же набор требований и т.д. В результате в нее будут загружены только уже имеющиеся в основной таблице записи. Записи, отвергнутые по другим причинам (кроме нарушения требования первичного ключа) несоответствия данных, в эту таблицу тоже не попадут.
3. Обновить данные, получающиеся при соединении реальной и рабочей таблицы.

Используя таблицу DEPT из предыдущих примеров и данные, полученные в результате последней загрузки (отделы 10, 20, 30, 40 и 50), загрузим следующие данные:

```
10,Sales,New  York,14-april-2001
60,Finance,Virginia,14-april-2001
```

В результате одна запись должна быть изменена, а одна — вставлена. Предполагая, что эти данные находятся в файле **new.dat**, создадим управляющий файл **load.ctl** следующего вида:

```
LOAD  DATA
INTO  TABLE  DEPT
APPEND
FIELDS  TERMINATED  BY  ','
(DEPTNO,
  DNAME          "upper(:dname)",
  LOC            "upper(:loc)",
  LAST_UPDATED   "My_to_date(:last_updated)"
)
```

Этот управляющий файл очень похож на предыдущий, но конструкции INFILE * и BEGINDATA удалены, а конструкция REPLACE заменена конструкцией APPEND. Файл, из которого надо загружать данные, мы будем задавать в командной строке, поэтому конструкция INFILE не нужна, а поскольку данные находятся во внешнем файле, то и BEGINDATA указывать не нужно. Так как мы хотим вставлять новые записи и изменять существующие, используется конструкция APPEND, а не REPLACE, как ранее. Итак, теперь можно загрузить данные с помощью команды:

```
C:\>sqlldr  userid=tkyte/tkyte  control=load.ctl  data=new.dat  errors=9999999
```

При выполнении этой команды будет сгенерирован BAD-файл с одной записью. Запись для отдела 10 окажется в файле new. bad, поскольку нарушает ограничение первичного ключа. Это можно проверить, обратившись к журнальному файлу, **load.log**:

```
Record 1: Rejected - Error on table DEPT.
ORA-00001: unique constraint (TKYTE.EMP_PK) violated
```

Теперь загрузим **BAD**-файл с помощью почти такого же управляющего файла. Надо изменить имя таблицы, в которую будут загружаться данные: **DEPT** на **DEPT_WORKING**, а также использовать конструкцию **REPLACE** вместо **APPEND**. Таблица, в которую будут загружаться данные, создается следующим образом:

```
tkyte@TKYTE816>  create  table  dept_working
  2  as
```

```
  3  select * from dept
  4   where 1=0
  5  /

Table created.

tkyte@TKYTE816> alter table dept_working
  2  add constraint dept_working_pk
  3  primary key(deptno)
  4  /

Table altered.
```

При загрузке данных не забудьте указать параметр **BAD=<имя файла>** в командной строке, чтобы не произошло чтения и записи одного и того же файла!

```
C:\sqlldr>sqlldr userid=tkyte/tkyte control=load_working.etl bad=working.bad
data=new.bad
```

После загрузки в таблице **DEPT_WORKING** окажется одна строка. Если в файле **WORKING.BAD** окажутся записи, значит, они действительно **плохие,** нарушают одно из требований целостности и требуют отдельного рассмотрения. Теперь, когда и эта загрузка выполнена, можно обновить существующие строки в таблице **DEPT** с помощью оператора **UPDATE:**

```
tkyte@TKYTE816> set autotrace on explain
tkyte@TKYTE816> update  (select /*+ ORDERED USE_NL(dept) */
  2                dept.dname         dept_dname,
  3                dept.loc           dept_loc,
  4                dept.last_updated  dept_last_updated,
  5                w.dname            w_dname,
  6                w.loc              w_loc,
  7                w.last_updated     w_last_updated
  8               from dept_working   W, dept
  9              where dept.deptno =  w.deptno)
 10     set dept_dname = w_dname,
 11         dept_loc    = w_loc,
 12         dept_last_updated = w_last_updated
 13  /

1 row updated.

Execution   Plan

    0       UPDATE STATEMENT Optimizer=CHOOSE  (Cost=83 Card=67 Bytes=5226)
    1    0   UPDATE OF  'DEPT'
    2    1     NESTED LOOPS  (Cost=83 Card=67 Byte3=5226)
    3    2       TABLE ACCESS  (FULL)  OF  'DEPT_WORKING'  (Cost=l Card=82
    4    2       TABLE ACCESS  (BY INDEX ROWID)  OF  'DEPT'  (Cost=l Card=8

    5    4         INDEX  (UNIQUE SCAN)  OF  'EMP_PK'  (UNIQUE)

tkyte@TKYTE816> select deptno, dname, loc, last_updated from dept;

 DEPTNO  DNAME          LOC          LAST_UPDA
-------  -------------- ------------ -----------------
     10  SALES          NEW YORK     14-APR-01
```

```
20 ACCOUNTING     VIRGINIA     13-APR-01
30 CONSULTING     VIRGINIA     14-APR-01
40 FINANCE        VIRGINIA     14-APR-01
50 FINANCE        VIRGINIA     02-APR-01
60 FINANCE        VIRGINIA     14-APR-01

6 rows selected.
```

Поскольку рабочая таблица обычно не анализируется, мы используем подсказку оптимизатору, предлагающую использовать таблицу **DEPT_WORKING** как ведущую при соединении. Необходимо, чтобы таблица **DEPT_WORKING** просматривалась полностью. Мы будем изменять строку, соответствующую каждой строке в этой таблице, а выбирать ее из таблицы **DEPT** для изменения будем по индексу (это указано в строках плана, следующих за **NESTED LOOPS). B** большинстве случаев такой подход наиболее эффективен.

Загрузка данных из отчетов

Иногда необходимо загрузить данные из текстового отчета. Эти данные представлены в определенном формате, но разбросаны по всему отчету. Например, мне пришлось загружать данные из отчета примерно такого вида:

```
3205679761 - Detailed Report

  July 01, 2000 21:24
    Location  :  location data 1
    Status    :  status 1

  July 01, 2000 22:18
    Location  :  location data 2
    Status    :  status 2


3205679783 - Detailed Report

  July 01, 2000 21:24
    Location  :  location data 3
    Status    :  status data 3
```

в следующую таблицу:

```
tkyte@TKYTE816>create table t
  2   (serial_no varchar2(20),
  3    date_time varchar2(50) ,
  4    location  varchar2(100) ,
  5     status     varchar2(100)
  6   )
  7   /

Table created.
```

Весьма вероятно, что с помощью ряда "хитроумных" триггеров, пакета PL/SQL для поддержки состояния и нескольких трюков **SQLLDR** этот отчет можно загрузить не-

посредственно. В руководстве *Oracle Server Utilities Guide* приводится пример такой загрузки. Однако мне этот подход кажется крайне сложным и непрозрачным, не говоря уже о том, что придется устанавливать триггеры для таблицы для поддержки процесса загрузки. А что если таблицы используются по ходу загрузки для других задач? Для них триггеры не должны срабатывать, но нет никакого способа создать триггер, срабатывающий только при вставке строк с помощью утилиты SQLLDR. Поэтому мне захотелось найти более простой способ.

Зачастую при загрузке сложных данных, как в представленном примере, быстрее, эффективнее и проще загрузить данные в рабочую таблицу, а затем обработать их с помощью небольшой хранимой процедуры. Именно так я и поступил с представленным выше отчетом. Я использовал следующий управляющий файл:

```
LOAD DATA
INTO TABLE TEMP
REPLACE
(seqno RECNUM,
 text Position(1:1024))
```

для загрузки данных в таблицу, созданную оператором

```
tkyte@TKYTE816> create table temp
  2   (seqno int primary key,
  3      text varchar2(4000))
  4   organization index
  5   overflow tablespace data;

Table created.
```

Конструкция RECNUM в управляющем файле требует от утилиты **SQLLDR** подставлять порядковый номер текущей записи для соответствующего столбца при загрузке данных. В результате первой записи будет присвоен номер 1, сотой — номер 100 и т.д. Затем я использовал маленькую хранимую процедуру для переформатирования данных в требуемый вид. Эта процедура последовательно читает каждую входную строку из таблицы. Затем она просматривает строку и:

- если строка содержит подстроку **Detailed Report,** выбирает из строки число и запоминает его в переменной **L_SERIAL_NO;**
- если строка содержит подстроку **Location,** данные о местонахождении помещаются в переменную **L_LOCATION;**
- если строка содержит подстроку **Status,** данные о состоянии помещаются в переменную **L_STATUS** и вставляется новая запись. В этот момент мы собрали все необходимые поля записи. Имеется порядковый номер, дата и время (см. следующий пункт списка) и местонахождение.
- для других строк мы проверяем, может ли строка быть преобразована в дату. Если не может, мы ее вообще пропускаем. Это делается в обработчике соответствующей исключительной ситуации.

Вот эта процедура:

```
tkyte@TKYTE816> create or replace procedure reformat
  2  as
  3        l_serial_no  t.serial_no%type;
  4        l_date_time  t.date_time%type;
  5        l_location   t.location%type;
  6        l_status     t.status%type;
  7        l_temp_date  date;
  8   begin
  9        for x in  (select * from temp order by seqno)
 10        loop
 11            if  (x.text like  '%Detailed Report%")  then
 12                 l_serial_no  := substr(x.text, 1,  instr(x.text,'-')-l) ;
 13            elsif  (x.text like  '%Location  :  %')  then
 14                 l_location  := substr(x.text,  instr(x.text,':')+2);
 15            elsif  (x.text like  '%Status %:%')  then
 16                 l_status  := substr(x.text,  instr(x.text,':')+2);
 17                 insert into t  (serial_no, date_time, location,  status)
 18                 values  (l_serial_no,  l_date_time,  l_location,  l_status);
 19             else
 20                  begin
 21                      l_temp_date  := to_date (ltrim(rtxim(x. text)),
 22                                               'Month dd,  yyyy hh24:mi');
 23                       l_date_time := x.text;
 24                  exception
 25                       when  others  then null;
 26                 end;
 27           end if;
 28        end loop;
 29 end;
 30  /

Procedure  created.
```

Если сравнить этот объем работы с тем, что необходимо было бы проделать для создания сложных триггеров, запоминающих состояние между вставками, и координации работы этих триггеров с другими приложениями, обращающимися к таблице, вывод очевиден. Загрузка непосредственно в целевую таблицу, может, и сэкономит пару операций ввода/вывода, но соответствующий код будет ненадежным в силу сложности и уровня применяемых приемов.

Из этого примера можно сделать вывод о том, что при решении сложной задачи надо стараться максимально уменьшить ее сложность. Один из способов добиться этого — использовать соответствующие задаче средства. В данном случае мы использовали PL/SQL для написания простой процедуры, помогающей преобразовать данные в нужный формат после загрузки. Это — самый простой способ решения задачи. Не всегда правильно делать все с помощью утилиты **SQLLDR,** однако во многих случаях утилита **SQLLDR** оказывается более подходящим средством, чем программы на PL/SQL. Используйте то, что лучше всего подходит для решения конкретной задачи.

Загрузка файла в поля типа LONG RAW или LONG

Хотя тип данных **LONG RAW** и не рекомендуется использовать в Oracle8i, иногда он встречается в старых приложениях, и работать с ним все равно приходится. Иногда необходимо загрузить файл или файлы, в столбец типа **LONG RAW,** и вместо того, чтобы писать для этого специальную программу, **хотелось бы** использовать утилиту **SQLLDR.** Хорошая новость в том, что это можно сделать, а плохая — что это не слишком просто и не совсем подходит для загрузки большого количества файлов.

Чтобы загрузить данные типа **LONG RAW** с помощью **SQLLDR,** в общем случае понадобится отдельный управляющий файл для каждого загружаемого файла (для каждой строки), если только все файлы не одинакового размера. Чтобы заставить утилиту **SQLLDR** это сделать, необходимо прибегнуть к хитрости. Придется работать с группой буферов размером 64 Кбайт или меньше и определить, сколько записей такого размера необходимо конкатенировать при загрузке. Допустим, требуется загрузить файл длиной 1075200 байт. Соответствующий управляющий файл может выглядеть так:

(Числа в скобках справа, выделенные наклонным шрифтом, не являются частью управляющего файла, они используются для ссылок на строки.)

```
options(bindsize=1075700, rows=l)          (1)
Load Data                                  (2)
Infile mydata.dat "fix 53760"              (3)
concatenate 20                             (4)
Preserve Blanks                            (5)
Into Table foo                             (6)
Append                                     (7)
(id constant l,bigdata raw(1075200))       (8)
```

Фокус в том, что 53760* 20 = 1075200, и 53760 — самое большое число, являющееся делителем 1075200, меньшим, чем 64 Кбайт. Необходимо найти самое большое целое число, не превышающее 64 Кбайт, для использования в качестве "фиксированного" размера записи, а затем конкатенировать 20 таких записей для получения одной физической записи — содержимого файла.

Итак, в строке (3) мы задали число 53760 как фиксированный размер входной записи. Это отключает обычную интерпретацию утилитой **SQLLDR** символа перевода строки как конца записи. Утилита **SQLLDR** теперь считает записью 53760 байт, независимо от содержащихся в них данных. Строка (4) указывает **SQLLDR,** что логическая запись (то, что загружается) будет состоять из 20 таких физических записей, соединенных вместе. Мы использовали параметр **bindsize=1075700** в строке (1), чтобы задать буфер связывания такого размера, чтобы хватило на весь входной файл, с запасом (для других столбцов). Наконец, в строке (8) мы задаем размер буфера для этого столбца типа **RAW** (стандартно используется буфер размером 255 байт).

Этот управляющий файл загрузит данные из файла **MYDATA.DAT** в таблицу **FOO,** помещая в столбец ID постоянное значение 1, а в столбец **BIGDATA** — содержимое

файла. Поскольку все это организовать непросто (найти самый большой делитель размера файла, не превышающий 64 Кбайт и т.д.), я написал небольшую переносимую программу на языке C, которая делает это автоматически. Мне часто приходится загружать содержимое файлов в столбцы типа **LONG** или **LONG RAW** и одновременно с этим заполнять другой столбец — первичный ключ. Поэтому я использую эту программу для создания управляющих файлов наподобие представленного выше. Она используется следующим образом:

```
genctl имя_файла имя_таблицы имя_длинного_столбца имя
    столбца_первичного_ключа значение_первичного_ключа RAW | CHAR
```

и для получения представленного выше управляющего файла я вызвал ее так:

```
genctl mydata.dat foo bigdata id 1 RAW > test.ctl
```

Программа определила размер файла **MYDATA.DAT,** вычислила размер физической записи и автоматически сгенерировала соответствующий управляющий файл. Исходный код программы **GENCTL** можно найти на Web-сайте издательства Wrox по адресу **http://www.wrox.com.**

Загрузка данных, содержащих символы новой строки

Загрузка данных, которые включают символ новой строки, всегда была проблематичной в **SQLLDR.** Символ новой строки является стандартным признаком конца записи для **SQLLDR,** и обходные пути загрузки в прежних версиях не давали требуемой гибкости. К счастью, в Oracle 8.1.6 и более поздних версиях появились новые варианты работы с такими данными.

Теперь загружать данные, содержащие символы новой строки, можно следующими способами:

- загружать данные, в которых вместо символа новой строки подставлена другая последовательность символов, представляющая новую строку (например, поместить в текст подстроку \n вместо символов новой строки), и заменять этот текст вызовом **CHR(10)** с помощью SQL-функции в ходе загрузки;
- использовать атрибут **FIX** директивы **INFILE** и загружать файл в виде записей фиксированной длины;
- использовать атрибут **VAR** директивы **INFILE** и загружать файл в виде записей переменной длины, в которых первые несколько байтов каждой записи содержат длину записи;
- использовать атрибут **STR** директивы **INFILE** и загружать файл в виде записей переменной длины, в которых конец записи представлен заданной последовательностью символов, а не символом новой строки.

Мы продемонстрируем все эти способы.

Использование другого символа вместо символа новой строки

Это — простой метод, если можно управлять генерацией загружаемых данных. Если данные можно легко преобразовать в требуемый формат при создании файла данных, этот способ прекрасно подходит. Идея в том, чтобы применять к данным SQL-функцию при загрузке в базу данных, заменяя некоторую строку символов символом новой строки. Давайте добавим еще один столбец в таблицу **DEPT:**

```
tkyte@TKYTE816> alter table dept add comments varchar2(4000);

Table altered.
```

Мы будем использовать этот столбец для загрузки в него текста. Вот как может выглядеть управляющий файл, содержащий встроенные данные:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ','
TRAILING NULLCOLS
(DEPTNO,
  DNAME        "upper(:dname) " ,
  LOC           "upper(:loo)",
  LAST_UPDATED "my_to_date(:last_updated)",
  COMMENTS     "replace(:comments, '\n', chr(10))"
)
BEGINDATA
10,Sales,Virginia,01-april-2001,This is the Sales\nOffice in Virginia
20,Accounting,Virginia,13/04/2001,This is the Accounting\nOffice in Virginia
30,Consulting,Virginia,14/04/2001,This is the Consulting\nOffice in Virginia
40,Finance,Virginia,987268297,This is the Finance\nOffice in Virginia
```

Хотелось бы обратить внимание на одну важную вещь. Представленный выше управляющий файл будет работать только на DOS-совместимых платформах, таких как Windows NT. На платформе UNIX необходимо использовать такое описание поля комментариев:

```
COMMENTS     "replace(:comments, '\\n', chr(10))"
```

Обратите внимание, что при вызове функции **replace** пришлось использовать \\n, а не просто \n, поскольку в UNIX утилита **SQLLDR** распознает \n как символ новой строки, а не строку из двух символов. Мы использовали конструкцию \\n для получения строковой константы \n в управляющем файле утилиты **SQLLDR** на платформе UNIX. При запуске утилиты **SQLLDR** с представленным выше управляющим файлом (при необходимости измененным соответствующим образом для работы в ОС UNIX), в таблицу **DEPT** будут загружены следующие данные:

```
tkyte@TKYTE816> select deptno, dname, comments from dept;
```

```
 DEPTNO DNAME          COMMENTS
------- -------------- ------------------------------
                       Office in Virginia

     20 ACCOUNTING     This is the Accounting
                       Office in Virginia

     30 CONSULTING     This is the Consulting
                       Office in Virginia

     40 FINANCE        This is the Finance
                       Office in Virginia
```

Использование атрибута FIX

Для решения задачи можно также использовать атрибут FIX. Для его использования загружаемые данные должны быть представлены в виде записей фиксированной длины. В каждой записи входного набора данных будет одно и то же количество байтов. Это особенно подходит для данных в фиксированных позициях. Именно эти данные обычно состоят из записей фиксированной длины. При использовании данных в "свободном формате", с разделителями, маловероятно, что записи будут иметь одинаковую длину (смысл использования данных с разделителями в том, чтобы каждая запись имела именно такой размер, какой необходим для размещения ее данных).

При использовании атрибута FIX необходимо указывать конструкцию INFILE, поскольку атрибут этот относится к конструкции INFILE. Кроме того, данные при этом должны находиться во внешнем файле, а не в управляющем. Итак, предполагая, что данные представлены в виде записей фиксированной длины, получаем следующий управляющий файл:

```
LOAD DATA
INFILE demo17.dat "fix 101"
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ' , '
TRAILING NULLCOLS
(DEPTNO,
  DNAME         "upper(:dname)",
  LOC           "upper(:loc)",
  LAST_UPDATED "my_to_date(:last_updated)",
  COMMENTS
)
```

Здесь указано, что файл данных будет состоять из записей длиной 101 байт. **В запись** включается завершающий символ новой строки, если он есть. В данном случае символ новой строки не имеет специального значения в файле данных. Это просто еще один символ, который надо (или не надо) загружать. Это надо понять — символ новой строки в конце записи (если он там есть) будет частью записи. Чтобы глубоко это осознать, необходима утилита для представления содержимого файла на экране в таком виде, чтобы можно было увидеть фактические данные. Чтобы такая утилита была переноси-

мой и работала на любой платформе, мы будем реализовывать ее средствами СУБД. Можно написать подпрограмму, которая будет использовать переменную типа **BFILE** для чтения файла операционной системы и выдавать его посимвольно на экран, показывая, где находятся символы возврата каретки (ASCII-код 13), перевода строки (ASCII-код 10), табуляции (ASCII-код 9) и другие специальные символы. На языке PL/SQL эта подпрограмма может выглядеть так:

```
tkyte@TKYTE816> create  or  replace
  2  procedure  file_dump (p_directory in varchar2,
  3                           p_filename   in varchar2)
  4  as
  5       type array is table of varchar2(5)  index by binary_integer;
  6
  7       l_chars   array;
  8       l_bfile   bfile;
  9    l_buffsize    number default 15;
 10       l_data    varchar2(30);
 11       l_len     number;
 12       l_offset number default 1;
 13       l_char    char(1);
 14  begin
 15       — специальные случаи - выдаем "управляющие последовательности",
          — чтобы было понятно
 16      l_chars(0)   := '\0';
 17      l_chars(13)  := '\r';
 18      l_chars(10) :== '\n';
 19      l_chars(9)   := '\t';
 20
 21      l_bfile  := bfilename(p_directory, p_filename);
 22      dbms_lob.fileopen(l_bflie);
 23
 24      l_len  := dbms_lob.getlength(l_bfile);
 25      while(l_offset < l_len)
 26      loop
 27           — выдаем смещение от начала файла в байтах
 28     dbms_output.put(to_char(l_offeet,'fm000000')       ||  '-' ||
 29                          to_char(l_offset+l_buffsize-1,  'fmOOOOOO')) ;
 30
 31           — теперь читаем BUFFSIZE байтов из файла для показа
 32           l_data  := utl_raw.cast_to_varchar2
 33            (dbms_lob.substr(l_bfile,             l_buffsize,  l_offset)) ;
 34
 35            — цикл по символам
 36          for i  in 1  . .  length(l_data)
 37           loop
 38               l_char  := substr(l_data,i,1);
 39
 40                — если символ — печатный,  просто выдаем его
 41               if ascii (l_char) between  32  and  126
 42               then
```

```
43                      dbms_output.put(lpad(l_char,3));
44                    — если это один из представленных выше специальных
                      —  символов,
45                    — выдаем вместо него заданный текст
46                    elsif  (l_chars.exists(ascii(l_char)))
47                    then
48              dbms_output.put(lpad(l_chars(ascii(l_char)),            3)) ;
49                    — если это двоичные данные, выдаем их в
                      — шестнадцатиричном виде
50                    else
51                         dbms_output.put(to_char(ascii(l_char),'0X'));
52                    end if;
53              end loop;
54              dbms_output.new_line;
55
56              l_offset  := l_offset + l_suffsize;
57        end loop;
58        dbms_lob.close(l_bfile);
59    end;
60    /

Procedure created.
```

Подробнее ***о пакете DBMS_LOB и данных типа BFILE можно прочитать*** *в приложении А в конце книги.*

Итак, если взять файл **данных следующего вида:**

```
tkyte@TKYTE816> host type demo17.dat
10,Sales,Virginia,01-april-2001,This is the Sales
Office in Virginia
20,Accounting,Virginia,13/04/2001,This is the Accounting
Office in Virginia
30,Consulting,Virginia,14/04/2001 12:02:02,This is the Consulting
Office in Virginia
40,Finance,Virginia,987268297,This is the Finance
Office in Virginia

tkyte@TKYTE816> exec  file_dump('MY_FILES',  'demo17.dat');
000001-000015   1 0   , S     a l e s ,     V i r g i n
000016-000030   i a , 0       1 - a p r i l - 2 0 0
000031-000045   1 ,  T h  i s     is      t h e      S
000046-000060   a l e s     \r\n O f f i c e        i n
000061-000075      V i r       g i n i a
000076-000090
000091-000105                                  \r \n 2  0  , A
000106-000120   c c o u         n t i n g , V i r g i
000121-000135   n i a ,       1 3 / 0     4 / 2 0 0 1 ,
000136-000150    T h i s        i s       t h e      A c e
000151-000165   o u n t       i n g    \r\n  O f f i c e
000166-000180       i n          V i r g i n i a
000181-000195
```

```
000196-000210                    \r\n3 0 , Con su
000211-000225 1 t ing , V i r g i n i a ,
000226-000240   1 4 / 0 4 / 2 0 0 1     1 2 : 0
000241-000255   2 : 0 2 , T h i s   is   th
000256-000270 e   Co   n s u l t i n g \ r \ n O
000271-000285 f f i c e   in   V i r g i n
000286-000300 i a
000301-000315     \r\n4 0 , F i n a n c e , V
000316-000330   i r g i n i a ,   9 8 7 2 6 8 2
000331-000345 9 7 , T h i s   is   the
000346-000360 F i n a n ce\r\n O f f i c e
000361-000375     in   V i r g i n i a
000376-000390
000391-000405                              \r \n

PL/SQL procedure successfully completed.
```

то с помощью этой утилиты можно убедиться, что все записи имеют длину 101 байт. Если посмотреть на строку данных, начинающуюся с **000091-000105,** в конце ее можно обнаружить последовательность (\r\n). Поскольку мы знаем, что последний символ в этой строке находится в позиции 105 от начала файла, то можем отсчитать символы обратно и убедиться, что символ \n имеет смещение 101. Далее в строке, начинающейся с **000196-000210,** мы по смещению 202 от начала файла видим еще один перевод строки, представляющий конец записи.

Теперь, зная точно, что все записи имеют длину 101 байт, мы готовы загружать их с помощью представленного выше управляющего файла с конструкцией **FIX 101.** В результате получим:

```
tkyte@TKYTE816> select "" || comments || '"' comments from dept;

COMMENTS

"This is the Sales
Office in Virginia


"This is the Accounting
Office in Virginia


"This is the Consulting
Office in Virginia


"This is the Finance
Office in Virginia
```

Обратите внимание, что каждая загруженная строка завершается символом новой строки — добавленная нами завершающая кавычка является первым символом в новой

строке (последним символом в поле **COMMENTS** является символ новой строки, поэтому кавычка и выдается в новой строке). Это произошло потому, что в каждой входной записи символ новой строки находится в позиции 101 и утилита **SQLLDR** не считает его разделителем записей. Если это не подходит, надо брать данные в кавычки и использовать конструкцию **OPTIONALLY ENCLOSED** BY, чтобы утилита **SQLLDR** загружала только данные в кавычках, без символа новой строки (не забудьте при этом добавить дополнительное место в каждой записи в атрибуте **FIX).** При этом завершающий символ новой строки не будет загружаться как часть данных. Итак, если изменить управляющий файл следующим образом:

```
LOAD DATA
INFILE demo18.dat "fix 101"
INTO TABLE DEFT
REPLACE
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
TRAILING NULLCOLS
(DEPTNO,
  DNAME          "upper(:dname)",
  LOC             "upper(:loc)",
  LAST_UPDATED "my_to_date(:last_updated)",
  COMMENTS
)
```

и изменить файл данных так, чтобы он имел следующий вид:

```
C:\sqlldr>TYPE demo18.dat
10,Sales,Virginia,01-april-2001,"This is the Sales
Office in Virginia"
20,Accounting,Virginia,13/04/2001,"This is the Accounting
Office in Virginia"
30,Consulting,Virginia,14/04/2001 12:02:02,"This is the Consulting
Office in Virginia"
40,Finance,Virginia,987268297,"This is the Finance
Office in Virginia"
```

с текстом в кавычках, он будет загружен следующим образом:

```
tkyte@TKYTE816> select '"' || comments || '"' comments from dept;

COMMENTS

"This is the Sales
Office in Virginia"

"This is the Accounting
Office in Virginia"

"This is the Consulting
Office in Virginia"

"This is the Finance
Office in Virginia"
```

Предупреждаю тех, кому повезло работать и в ОС Windows NT, и в ОС UNIX. На этих платформах признаки конца строки различны. В ОС UNIX это просто \n. В ОС

Windows NT это \r\n. Предположим, мы передали файлы из предыдущего примера с помошью ftp-клиента на UNIX-машину. Если выполнить затем процедуру **FILE_DUMP,** можно понять, с какой проблемой придется столкнуться:

```
ops$tkyte@ORA8I.WORLD> EXEC file_dump('MY_FILES', 'demo17.dat');
000001-000015    1 0 , S a l e s , V i r g i n
000016-000030    i a , 0 1 - a p r i l - 2 0 0
000031-000045  1 , T h i s  is  the  S
000046-000060   a l e s \n O f f i c e  in
000061-000075    V i r g i n i a
000076-000090
000091-000105                          \n 2 0 , A c e
000106-000120    o u n t i n g , V i r g i n i
000121-000135   a , 1 3 / 0 4 / 2 0 0 1 , T h
000136-000150  is  is  the  A c c o u
000151-000165   n t i n g \ n O f f i c e  in
000166-000180      V i r g i n i a
000181-000195
000196-000210          \n3 0 , C o n s u l t i n
000211-000225   g , V i r g i n i a , 1 4 / 0
000226-000240  4 / 2 0 0 1  1 2 : 0 2 : 0 2
000241-000255  , T h i s  is  the  Co
000256-000270   n s u l t i n g \ n  O f f i c e
000271-000285    in  V i r g i n i a
000286-000300                          \n 4 0
000301-000315    P i n a n c e , V i r g i n i
000316-000330   a , 9 8 7 2 6 8 2 9 7 , T h i
000331-000345  s  i s  t h e  F i n a n c
000346-000360   e \ n O f f i c e  in  Vir
000361-000375  g i n i a
000376-000390
000391-000405          \n

PL/SQL procedure successfully completed.
```

Файл имеет другой размер, как и записи в нем. Каждая пара \r\n теперь представлена символом \n. В этом примере достаточно изменить значение атрибута **FIX** с 101 на 99, но **только** потому, что в каждой записи одинаковое количество строк! Длина каждой записи просто сократилась на 2 байта. Если бы в некоторых записях было три строки, их длина сократилась бы на три, а не на два байта. При этом файл изменился бы так, что записи не имели бы больше фиксированной длины. При использовании атрибута **FIX** убедитесь, что файл **создается и загружается** на однородных платформах (UNIX и UNIX, или Windows и Windows). При передаче файлов с одной из этих систем на другую они наверняка перестанут загружаться.

Использование атрибута VAR

Еще один метод загрузки данных со встроенными символами новой строки — с помощью атрибута **VAR.** При использовании этого формата каждая запись начинается с фиксированного количества байтов, в которых указана ее суммарная длина. С помощью

этого формата можно загружать записи переменной длины, содержащие символы новой строки, но только если в начале **каждой записи** указана **ее длина.** Итак, если использовать управляющий файл следующего вида:

```
LOAD DATA
INFILE demo19.dat "var 3"
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ','
TRAILING NULLCOLS
(DEPTNO,
  DNAME          "upper(:dname)",
  LOC            "upper(:loc)",
  LAST_UPDATED   "my_to_date(:last_updated)",
  COMMENTS
)
```

то атрибут **var** 3 будет означать, что первые три байта каждой входной записи содержат ее длину. Если взять файл данных следующего вида:

```
C:\sglldr>type demo19.dat
07110,Sales,Virginia,01-april-2001,This is the Sales
Office in Virginia
0762:0,Accounting,Virginia,13/04/2001,This is the Accounting
Office in Virginia
08730,Consulting,Virginia,14/04/2001 12:02:02,This is the Consulting
Office in Virginia
07140,Finance,Virginia,987268297,This is the Finance
Office in Virginia
```

то их можно успешно загрузить с помощью представленного управляющего файла. В этом файле загружаемых данных четыре записи. Первая запись начинается с **071,** т.е. следующим 71 байт представляет первую запись. В эти 71 байт входит и завершающий символ новой строки после слова **Virginia.** Следующая запись начинается с **078.** В ней 78 байт, и так далее. Используя файл данных такого формата, очень легко загрузить данные со встроенными символами новой строки.

И в этом случае, если приходится работать с UNIX и NT (показан пример для NT, где новая строка представлена двумя символами), надо уточнять длину каждой записи. В ОС UNIX представленный выше файл данных придется изменить, задав 69, 76, 85 и 69 в качестве длины записей.

Использование атрибута STR

Это, наверное, самый гибкий метод загрузки данных, содержащих символы новой строки. С помощью атрибута **STR** можно задать новую последовательность символов, представляющую конец строки. Это позволяет создать файл данных, в котором конец каждой записи помечен какими-либо специальными символами, и символ новой строки при этом теряет свое значение.

Я предпочитаю использовать несколько символов, специальный маркер, а затем символ новой строки. Это упрощает поиск конца записи при просмотре файла данных в

текстовом редакторе или с помощью утилит командной строки, поскольку каждая запись по-прежнему завершается символом новой строки. Атрибут **STR** задается в шестнадцатеричном виде, а автоматически преобразовать строку в шестнадцатеричный вид проще всего с помощью SQL и пакета **UTL_RAW** (подробнее о пакете **UTL_RAW** см. в приложении о стандартно поставляемых пакетах в конце книги). Например, при работе в Windows NT, где признаком конца строки является последовательность **CHR(13) || CHR(10)** (возврат каретки/перевод строки), а в качестве специального маркера выбран символ конвейера, |, можно преобразовать строку так:

```
tkyte@TKYTE816> select  utl_raw.cast_to_raw('|'||chr(13)||chr(10))  from
dual;


UTL_RAW.CAST_TO_RAW('|'||CHR(13)||CHR(10))

7C0D0A
```

В качестве значения атрибута **STR** надо задать **X'7C0D0A".** Для загрузки можно создать управляющий файл следующего вида:

```
LOAD DATA
INFILE demo20.dat  "str X'7C0D0A'"
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY
TRAILING NULLCOLS
 (DEPTNO,
   DNAME        "upper(:dname) " ,
   LOC           "upper(:loc)",
   LAST_UPDATED "my_to_date(:last_updated)",
   COMMENTS
)
```

Если загружаемые данные имеют следующий вид:

```
C:\sqlldr>type  demo20.dat
10,Sales,Virginia,01-april-2001,This is the Sales
Office in Virginia|
20,Accounting,Virginia,13/04/2001,This is the Accounting
Office in Virginia|
30,Consulting,Virginia,14/04/2001 12:02:02,This is the Consulting
Office in Virginia|
40,Finance,Virginia,987268297,This is the Finance
Office in Virginia|
```

так что каждая запись данных завершается последовательностью **|\r\n**, то с помощью представленного выше управляющего файла их можно корректно загрузить.

Как упростить обработку символов новой строки

Итак, мы рассмотрели четыре способа загрузки данных со встроенными символами новой строки. В следующем разделе мы используем один из них, атрибут **STR,** при со-

здании универсальной утилиты выгрузки данных, полностью решающей проблему с символами новой строки в текстах.

Кроме того, всегда надо учитывать, как я уже упоминал в предыдущих примерах, что в Windows (во всех версиях), текстовые файлы могут завершаться символами \r\n (ASCII-код 13 + ASCII-код 10, возврат каретки /перевод строки). В управляющем файле (в задаваемом для атрибутов **FIX** и **VAR** количестве байтов, а также в строке значения для атрибута **STR)** должно быть учтено, что символ \r является частью записи. Например, если взять любой из представленных выше файлов данных, содержащих символы \r\n, и передать их по протоколу FTP на ОС UNIX в стандартном текстовом (ASCII) режиме, каждая пара символов \r\n будет преобразована в \n. Управляющий файл, только что работавший в Windows, уже не позволит загрузить данные. Об этом надо всегда помнить при создании управляющего файла.

Выгрузка данных

Утилита **SQLLDR,** да и вообще ни одно из поставляемых в составе СУБД Oracle инструментальных средств, не позволяет выгрузить данные в формате, подходящем для загрузки с помощью **SQLLDR.** Это можно использовать при переносе данных из одной системы в другую без применения утилит **EXP/IMP**. Утилиты **EXP/IMP** позволяют эффективно переносить средние объемы данных из одной системы в другую. Поскольку утилита **IMP** не обеспечивает непосредственный импорт и не позволяет распараллелить построение индексов, перенос данных с помощью **SQLLDR** с последующим параллельным созданием индексов без журнализации может выполняться на несколько порядков быстрее.

Создадим небольшую утилиту на языке PL/SQL, которую можно будет использовать для выгрузки данных на сервере в формате, подходящем для загрузки с помощью **SQLLDR.** Кроме того, аналогичные средства на базе **Pro*C** и утилиты SQL*Plus представлены на Web-сайте издательства Wrox. Утилита на PL/SQL подходит для большинства случаев, но реализация на **Pro*C** обеспечивает более высокую производительность, а также пригодится, если необходимо сгенерировать файлы данных на клиенте (а не на сервере, где их будет генерировать PL/SQL-утилита).

Создаваемый пакет имеет следующую спецификацию:

```
tkyte@TKYTE816> create or replace package unloader
  2  as
  3  function run(p_query in varchar2,
  4     p_tname           in varchar2,
  5     p_mode            in varchar2 default 'REPLACE',
  6     p_dir             in varchar2,
  7     p_filename        in varchar2,
  8     p_separator      in varchar2 default ' , ' ,
  8     p_enclosure       in varchar2 default "",
 10     p_terminator      in varchar2 default '|')
 11  return number;
 12  end;
 13  /

Package created.
```

Параметры функции имеют следующее назначение:

```
/* Функция run — выгружает результаты запроса в файл и создает
                управляющий файл для загрузки этих данных в другую
                таблицу
   p_query     = SQL-запрос для "выгрузки". Запрос может быть практически
                любым.
   p_tname     = Таблица, в которую надо загружать данные. Это имя будет
                указано в управляющем файле.
   p_mode      = REPLACE|APPEND|TRUNCATE — способ загрузки данных
   p_dir       = Каталог, в который будут записаны файлы .ctl и .dat.
   p_filename  = Имя файла, в который надо записывать. Я добавляю к
                этому имени суффиксы .ctl и .dat
   p_separator = Разделитель полей, по умолчанию — запятая.
   p_enclosure = Символ, в который заключается каждое поле.
   p_terminator= Признак конца строки. Мы будем использовать его, чтобы
                можно было выгружать и затем загружать данные со
                встроенными символами новой строки. Стандартное
                значение — '|\n' (символы конвейера и перевода строки);
                для NT можно использовать '|\r\n'. Это значение надо
                изменять, только если точно известно, что оно есть в
                загружаемых данных. Я всегда добавляю в конце
                последовательности символы конца строки для
                соответствующей ОС, но это не обязательно.
*/
```

Тело пакета представлено ниже. Для создания управляющего файла и файла данных используется пакет **UTL_FILE.** Не забудьте почитать о настройке пакета UTL_FILE в Приложении А. Если не установить соответствующим образом параметр в файле **init.ora,** пакет **UTL_FILE** работать не будет. Для динамического выполнения любого запроса используется пакет **DBMS_SQL** (подробнее о нем см. в соответствующем разделе приложения). В запросах используется один тип данных — **VARCHAR2(4000).** Поэтому этот метод нельзя использовать для выгрузки больших объектов размером более 4000 байт. Но выгружать до 4000 байт любого большого объекта с помощью функции **DBMS_LOB.SUBSTR** все же можно. Кроме того, поскольку в качестве единственного типа результатов используется **VARCHAR2,** мы можем обрабатывать столбцы типа **RAW** длиной до 2000 байт (4000 шестнадцатиричных цифр) — этого вполне достаточно для всех данных, кроме данных типа **LONG RAW** и больших двоичных объектов. Представленный ниже пакет решает проблему в 90 процентах случаев. Немного потрудившись и используя другие средства, описанные в книге, например пакет **LOB_IO,** представленный в главе 18, можно расширить его для поддержки всех типов данных, в том числе больших объектов любого размера.

```
tkyte@TKYTE816> create or replace package body unloader
  2  as
  3
  4
  5  g_theCursor          integer default dbms_sql.open_cursor;
  6  g_descTbl          dbms_sql.desc_tab;
  7  g_nl               varchar2(2) default chr(10);
  8
```

Это объявления ряда глобальных переменных, которые используются в пакете. Глобальный курсор открывается один раз, при первом обращении к пакету, и остается открытым до завершения сеанса. Это избавляет от необходимости открывать курсор при каждом обращении к пакету. **G_DESCTBL** — это PL/SQL-таблица, в которую будут попадать результаты вызова **DBMS_SQL.DESCR1BE. G_NL** — это последовательность, представляющая новую строку. Мы будем использовать ее в строках, где надо оставить встроенные символы перевода строк. Для работы в Windows ничего настраивать не надо — подпрограммы пакета UTL_FILE распознают символ **CHR(10)** в переданной строке и автоматически преобразуют его в последовательность возврат каретки/перевод строки.

Далее идет простая функция, используемая для преобразования символа в шестнадцатеричный вид. Для этого используются встроенные функции:

```
 9
10  function to_hex(p_str in varchar2)  return varchar2
11  is
12  begin
13       return to_char(ascii(p_str),  'fm0x');
14  end;
15
```

Ниже представлена процедура создания управляющего файла для загрузки выгруженных данных. Она использует для этого таблицу **DESCRIBE,** генерируемую вызовом **dbms_sql.describe_columns.** Процедура учитывает особенности ОС, в частности то, использует ли ОС последовательность возврат каретки/перевод строки (это учитывается при задании атрибута **STR)** и какой символ является разделителем каталогов в именах файлов: \ или /. Для этого просматривается переданное при вызове имя каталога. Если это имя содержит символ \, значит, мы работаем в ОС Windows, иначе — в ОС UNIX:

```
16  /*
17  */
18
19  procedure   dump_ctl(p_dir          in varchar2,
20                       p_filename     in varchar2,
21                       p_tname        in varchar2,
22                       p_mode         in varchar2,
23                       p_separator    in varchar2,
24                       p_enclosure    in varchar2,
25                       p_terminator   in varchar2)
26  is
27       l_output        utl_file.file_type;
28       l_sep           varchar2(5);
29       l_str           varchar2(5);
30       l_path          varchar2(5);
31  begin
32      if  (p_dir like  '%\%')
33       then
34             - Windows platforms -
35          l_str  := chr(13)  || chr(10);
```

```
36            if  (p_dir not like '%\' AND p_filename not like  '\%')
37            then
38                 l_path  :=  '\';
39            end if;
40       else
41            l_str  := chr(10);
42           if  (p_dir not like  '%/'  AND p_filename not like  '/%')
43            then
44                 l_path  :=  '/';
45            end if;
46       end if;
47
48      l_output  := utl_file.fopen(p_dir, p_filename || '.ctl', 'w');
49
50      utl_file.put_line(l_output,  'load data');
51      utl_file.put_line(l_output,  'infile ''' || p_dir || l_path  ||
52                                  p_filename  ||  '.dat"  "str x'''  ||
53                                   utl_raw.cast_to_raw(p_terminator  ||
54                                                  l_str  )  ||  '''');
55     utl_file.put_line(l_output,  'into table  '  || p_tname);
56      utl_file.put_line(l_output, p_mode);
57      utl_file.put_line(l_output,  'fields  terminated  by  x'''  ||
58                                   to_hex(p_separator)  ||
59                  '''                    enclosed  by  X'''  ||
60                                 to_hex (p_enclosure)  ||  '''  ' ) ;
61     utl_file.put_line(l_output,  '(');
62
63      for  i  in  1  . .  g_descTbl.count
64       loop
65            if  (g_descTbl(i).col_type  =  12  )
66            then
67                 utl_file.put(l_output,  l_sep  ||  g_descTbl(i).col_name  ||
68                                        '  date  ''ddmmyyyyhh24miss''  ' ) ;
69            else
70                  utl_file.put(l_output,  l_sep  ||  g_descTbl(i).col_name ||
71                             '  char('  ||
72                            to_char(g_descTbl(i).col_max_len*2)  || ' )');
73            end if;
74          l_sep  :=  ' , ' ||g_nl;
75       end  loop;
76      utl_file.put_line(l_output, g_nl  ||  ') ') ;
77       utl_file.fclose(l_output);
78  end;
```

Вот простая функция, выдающая переданную строку в заданных "кавычках". Обратите внимание, что она не просто берет строку в кавычки, но и удваивает все символы кавычек, входящие в строку, так что они сохраняются при загрузке:

```
79
80  function quote(p_str  in varchar2,  p_enclosure  in varchar2)
81              return varchar2
82  is
```

```
 83  begin
 84       return p_enclosure  ||
 85                replace(p_str, p_enclosure, p_enolosure||p_enclosure)  ||
 86               p_endosure;
 87  end;
```

Теперь переходим к основной функции, RUN. Поскольку она достаточно большая, комментарии будут делаться по ходу:

```
 88
 89  function run(p_query       in varchar2,
 90               p_tnaroe      in varchar2,
 91               p_mode        in varchar2 default  'REPLACE',
 92               p_dir         in varchar2,
 93               p_filename    in varchar2,
 94              p_separator   in varchar2 default  ',' ,
 95              p_enclosure   in varchar2 default  '"' ,
 96               p_terminator  in varchar2 default '|') return number
 97  is
 98    l_output     utl_file.file_type;
 99   l_columnValue varchar2 (4000) ;
100    l_colCnt     number default 0;
101   l_separator  varchar2(10) default ' ';
102    l_cnt        number default 0;
103    l_line       long;
104    l_datefmt    varchar2(255) ;
105    l_descTbl    dbms_sql.desc_tab;
106  begin
```

Текущий формат даты мы сохраним в переменной, чтобы можно было заменить его форматом, сохраняющим дату и время при сбросе данных на диск. Таким образом, мы будем сохранять время суток при выдаче дат. Затем зададим обработчик исключительных ситуаций, чтобы восстанавливать значение NLS_DATE_FORMAT в случае ошибки:

```
107       select value
108          into l_datefmt
109          from nls_session_parameters
110        where parameter =  'NLS_DATE_FORMAT';
111
112       /*
113           Устанавливает формат даты а виде большой строки цифр.  Тем
114           самим снимаются все проблемы NLS и сохраняется не только
              дата, но и время.
115       */
116       execute immediate
117           'alter session set nls_date_format=''ddmmyyyyhh24miss''';
118
119       /*
120           Создаем блок с обработчиком исключительных ситуаций, чтобы в
121           случае ошибки можно было восстановить формат даты.
122       */
123       begin
```

Теперь разберем и опишем запрос. Присваивание переменной **G_DESCTBL** значения **L_DESCTBL** делается для "сброса" глобальной таблицы, иначе, кроме данных текущего запроса, она может содержать результаты предыдущего вызова **DESCRIBE.** После этого вызываем процедуру **DUMP_CTL** для создания управляющего файла:

```
124          /*
125              Разбираем и описываем запрос.  Присваиваем
126              descTbl пустую таблицу, так что ее атрибут  .count будет
127              давать правильное значение.
128          */
129          dbms_sql.parse(g_theCursor, p_query, dbms_sql.native);
130      g_descTbl       := l_desoTbl;
131          dbms_sql.describe_columns(g_theCursor, l_colCnt, g_descTbl);
132
133          /*
134              Создаем управляющий файл для загрузки этих данных
135              а указанную таблицу.
136          */
137          dump_ctl(p_dir, p_filename, p_tname, p_mode, p_separator,
138                                  p_endosure, p_terminator);
139
140          /*
141             Связываем каждый столбец с varchar2(4000) . Нас не
142             интересует, выбирается ли число, дата или данные другого
143             типа. Все может быть неявно преобразовано в строку.
144          */
```

Теперь все готово для сброса данных на диск. Начнем с объявления каждого выбираемого столбца как имеющего тип **VARCHAR2(4000).** Данные типа NUMBER, DATE, RAW и всех остальных типов будут преобразовываться в **VARCHAR2.** Сразу после этого выполним запрос, чтобы подготовиться к выборке данных:

```
145          for i in 1 .. l_colCnt loop
146            dbms_sql.define_column(g_theCursor, i, l_columnValue, 4000);
147          end loop;
148
149          /*
150              Выполняем запрос, игнорируя результаты вызова execute.
151              Они имеют смысл только для операторов вставки, изменения
                 или удаления.
152          */
```

Открываем файл данных для записи, выбираем все строки результатов запроса и выдаем их в файл данных:

```
153          l_cnt := dbms_sql.execute(g_theCursor);
154
155          /*
156              Открываем файл для записи результатов и выдаем их туда
157              через разделитель.
158          */
159          l_output := utl_file.fopen(p_dir, p_filename || '.dat', 'w' ,
```

```
160                                                    32760) ;
161             loop
162                 exit when (dbms_sql.fetch_rows(g_theCursor) <= 0) ;
163                  l_separator := '';
164                  l_line := null;
165                 for i in 1 .. l_colCnt loop
166                      dbms_sql.column value(g theCursor, i,
167                                                  l_columnValue);
168                     l_line := l_line || l_separator ||
169                                  quote(l_columnValue, p_enclosure);
170                      l_separator := p_separator;
171                  end loop;
172                 l_line := l_line || p_terminator;
173                 utl_file.put_line(l_output, l_line);
174                 l_cnt := l_cnt+1;
175             end loop;
176            utl_file.fclose(l_output);
```

Наконец, мы восстанавливаем формат даты (обработчик исключительных ситуаций сделает то же самое, если в представленном выше коде будет любая ошибка) и завершаем работу:

```
177
178             /*
179                 Теперь восстанавливаем формат даты и возвращаем
180                количество строк, записанных в файл данных.
181             */
182             execute immediate
183               'alter session set nls_date_format=''' || l_datefmt || '''';
184             return l_cnt;
185          exception
186             /*
187                 В случае ЛЮБОЙ ошибки восстанавливаем формат даты и
188                 повторно возбуждаем исключительную ситуацию.
189             */
190             when others then
191             execute immediate
192            'alter session set nls_date_format=''' || l_datefmt || '''';
193             RAISE;
194       end;
195   end run;
196
197
198   end unloader;
199   /

Package body created.
```

Теперь эту функцию можно использовать так, как показано ниже.

Для выполнения следующих действий, естественно, необходимо предоставить текущему пользователю или роли привилегию SELECT для таблицы SCOTT.EMP.

```
tkyte@TKYTE816> drop  table  emp;

Table  dropped.

tkyte@TKYTE816> create table emp as select * from scott.emp;

Table  created.

tkyte@TKYTE816> alter table emp add resume raw(2000);

Table  altered.

tkyte@TKYTE816> update  emp
  2       set resume = rpad('02', 4000, '02');

14  rows updated.

tkyte@TKYTE816> update  emp
  2       set ename = substr(ename, 1, 2)  ||  '"'  ||
  3                       chr(10)  ||  '"'  ||  substr(ename,3);

14  rows updated.

tkyte@TKYTE816> set serveroutput on
tkyte@TKYTE816> declare
  2        l_rows     number;
  3   begin
  4        l_rows  := unloader.run
  5                    (p_query       =>  'select * from emp order by erapno',
  6                    p_tname        =>  'emp',
  7                    p_mode         =>  'replace',
  8                    P_dir          =>  'c:\temp',
  9                    p_filename     =>  'emp' ,
 10                    p_separator    =>   ',',
 11                    p_enclosure    =>  '"' ,
 12                    p_terminator   =>  '~');
 13
 14       dbms_output.put_line(to_char(l_rows)  ||
 15                                    ' rows extracted to ascii file');
 16   end;
 17 /
14  rows extracted to ascii  file

PL/SQL procedure  successfully  completed.
```

Вот какой управляющий файл будет сгенерирован в результате.

Чисел в круглых скобках справа, выделенных наклонным шрифтом, на самом деле в файле нет. Они используются для ссылок в тексте:

```
load data                                                    (1)
infile 'c:\temp\emp.dat' "str      x'7E0D0A"'                (2)
into table emp                                               (3)
replace                                                      (4)
fields terminated by X'2c' enclosed by X'22'                 (5)
```

```
(                                                    (6)
EMPNO char(44) ,                                     (7)
ENAME char(20) ,                                     (B)
JOB  char(18),                                       (9)
MGR char(44) ,                                       (lO)
HIREDATE date  'ddmmyyyyhh24miss',                   (11)
SAL  char(44),                                       (12)
COMM char(44) ,                                      (13)
DEPTNO  char(44),                                    (14)
RESUME char(4000)                                    (15)
)                                                    (16)
```

Хотелось бы обратить внимание на следующее.

- Строка (2). Используется новое средство Oracle 8.1.6, атрибут STR. Можно указать, какой символ или какая строка используется как признак конца записи. Это позволяет загружать данные со встроенными символами перевода строк. Указанная выше строка x'7E0D0A' — это просто тильда, за которой идут символы новой строки.
- Строка (5). Используются заданные разделители и кавычки. Я не использую конструкцию OPTIONALLY ENCLOSED BY, поскольку в кавычки, после удвоения соответствующих символов в данных, будет заключаться каждое поле.
- Строка (11). Используется длинный "числовой" формат даты. Последствия этого двояки. Во-первых, решаются проблемы NLS для дат, а во-вторых, в полях дат сохраняется компонент времени.
- Строка (15). Используются данные типа char(4000). По умолчанию утилита SQLLDR будет использовать для полей тип данных char(255). Я удвоил длину каждого поля в управляющем файле. Это гарантирует, что утилита SQLLDR сможет прочитать данные без усечения полей. Удвоение сделано для поддержки данных типа RAW, которые мы получаем в формате VARCHAR2 (в шестнадцатеричном виде). Т.е. каждый байт исходных данных представляется двумя символами.

Представленный выше код генерирует следующий файл данных (с расширением **.dat**):

```
"7369","SM""
,""ITH",CLERK","7902","17121980000000","800","","20","02020202020202020202020202
0202020202020202020...<существенная часть удалена>  ...0202020202020202"~
"7499","AL""
""LEN","SALESMAN","7698","20021981000000","1600","300","30","020202020202020202
0202020202020202020...Существенная часть удалена>  ...0202020202020202"~
```

Обратите внимание на следующее в этом файле данных.

- Каждое поле взято в указанные кавычки.
- Поле ENAME, в которое мы с помощью оператора UPDATE внесли кавычки и символ \n, содержат встроенный перевод строки. Кроме того, символы кавычек удвоены, чтобы они остались в данных после загрузки.
- Данные типа DATE выгружены в виде больших чисел.

- Добавленные поля типа **RAW** сохранились и выданы в шестнадцатиричном виде.
- Каждая запись в файле завершается тильдой (~), как и требовалось.

Теперь легко загрузить эти данные с помощью утилиты **SQLLDR.** При вызове **SQLLDR** можно добавить необходимые опции командной строки.

Эту функцию можно использовать для решения задач, которые сложно или невозможно решить по-другому. Например, если необходимо переименовать столбец ENAME в таблице **EMP** в **EMP_NAME,** можно сделать так, как показано ниже. Выгружаем данные для последующей загрузки с помощью **SQLLDR.** Обратите внимание, как мы "переименовываем" столбец, задав псевдоним **EMP_NAME** в списке выбора оператора SELECT после **ENAME**. В результате управляющий файл будет создан со столбцом **EMP_NAME** вместо **ENAME**. Затем я удаляю данные из таблицы, удаляю неправильно названный столбец и добавляю столбец с новым именем. После этого данные загружаются обратно в таблицу. Этот подход предпочтительней по сравнению с более простым: "добавить столбец, изменить его данные, взяв их из старого, удалить старый столбец", если таблицу надо коренным образом реорганизовывать или объем данных в сегментах отката и журнале повторного выполнения, генерируемых при подобной операции, имеет значение. Можно вызвать утилиту **SQLLDR** в режиме непосредственной загрузки и вообще избежать генерации данных отката и повторного выполнения при загрузке. Вот как это делается, по шагам:

```
tkyte@TKYTE816> declare
  2        l_rows     number;
  3  begin
  4       l_rows  := unloader.run
  5                    (p_query        => 'select EMPNO, ENAME EMP_NAME,
  6                                              JOB , MGR, HIREDATE,
  7                                               SAL, COMM, DEPTNO
  8                                          from emp
  9                                        order by empno' ,
 10             p_tname               => 'emp' ,
 11                      p_mode        => 'TRUNCATE',
 12                      P_dir         => 'o:\temp',
 13                      p_filename    => 'emp',
 14                      p_separator   =>  '.'
 15                      p_enclosure   => '"',
 16                      p_terminator =>' ~ ' ) ;
 17
 18       dbms_output.put_line(to_char(l_rows) ||
 19                                  ' rows extracted to ascii file');
 20  end;
 21  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> truncate table emp;

Table truncated.

tkyte@TKYTE816> alter table emp drop column ename;
```

```
Table altered.

tkyte@TKYTE816> alter table emp add emp_name varchar2(10) ,-

Table altered.

tkyte@TKYTE816> host sqlldr userid=tkyte/tkyte control=c:\temp\emp.ctl
SQLLDR: Release 8.1.6.0.0 - Production on Sat Apr 14 20:40:01 2001
(c) Copyright 1999 Oracle Corporation.  All rights reserved.
Commit point reached - logical record count 14

tkyte@TKYTE816> desc emp
 Name                    Null?      Type

 EMPNO                   NOT NOLL   NUMBER(4)
 JOB                                VARCHAR2(9)
 MGR                                NUMBER(4)
 HIREDATE                           DATE
 SAL                                NUMBER(7,2)
 COMM                               NUMBER(7,2)
 DEPTNO                             NUMBER(2)
 RESUME                             RAH(2000)
 EMP_NAME                           VARCHAR2(10)

tkyte@TKYTE816> select emp_name from emp;

EMP NAME

SM"
"ITH

MI"
"LLER

14 rows selected.
```

Этот метод можно использовать и для выполнения других операций, скажем, для изменения типа данных, их денормализации (выгрузки результатов соединения, например) и т.д.

Как уже было сказано, представленный выше алгоритм работы утилиты выгрузки можно реализовать на различных языках и с помощью разных средств. На Web-сайте издательства Wrox можно найти реализацию этого алгоритма не только на PL/SQL, но и на языке Pro*C и в виде сценариев SQL*Plus. Реализация алгоритма на Pro*C будет самой быстродействующей и всегда будет создавать файлы в файловой системе клиента. Реализация алгоритма на PL/SQL — наиболее универсальна (ничего не надо компилировать и устанавливать на клиентских рабочих станциях), но файлы будут всегда создаваться в файловой системе сервера. Реализация алгоритма в виде сценариев SQL*Plus является компромиссным вариантом, обеспечивая прекрасную производительность и возможность создавать файлы на клиентской машине.

Загрузка больших объектов

Давайте рассмотрим некоторые методы загрузки данных в большие объекты. Речь идет не о столбцах типа **LONG** или **LONG RAW,** а о более предпочтительных типах **BLOB** и **CLOB.** Эти типы данных появились в Oracle 8.0 и предоставляют намного больше возможностей для работы, чем устаревшие типы **LONG** и **LONG RAW.**

Мы изучим два метода загрузки данных в эти столбцы — с помощью **SQLLDR** и процедур на PL/SQL. Существуют и другие, например потоки Java, программирование на Pro*C и использование непосредственно функционального интерфейса **OCI.** В главе 18 рассматривается, как выгружать большие объекты с помощью Pro*C. Загрузка большого объекта будет выполняться аналогично, но вместо вызова **EXEC SQL READ** придется использовать **EXEC SQL WRITE.**

Начнем с рассмотрения метода загрузки больших объектов с помощью PL/SQL, а затем опишем, как это сделать с помощью утилиты **SQLLDR.**

Загрузка больших объектов с помощью PL/SQL

Пакет **DBMS_LOB** содержит подпрограмму **LOADFROMFILE.** Эта процедура позволяет использовать данные типа **BFILE** (представляющие файл в операционной системе) для наполнения столбцов типа **BLOB** или **CLOB** в базе данных. Чтобы можно было воспользоваться этой процедурой, необходимо создать в базе данных объект **DIRECTORY.** Этот объект позволит создавать объекты типа **BFILE** (и открывать их), ссылающиеся на файлы, существующие в файловой системе, к которой имеет доступ сервер баз данных. Последнее уточнение ("... к которой имеет доступ сервер баз данных") — ключевой момент при использовании PL/SQL для загрузки больших объектов. Пакет **DBMS_LOB** работает на сервере. Он может обращаться только к тем файловым системам, к которым имеет доступ сервер. В частности, для пакета недоступна локальная файловая система рабочей станции, с которой вы обращаетесь к СУБД Oracle по сети. Нельзя с помощью PL/SQL загрузить большие объекты непосредственно с локальной машины, поскольку их нет на сервере.

Итак, надо начать с создания объекта **DIRECTORY** в базе данных. Сделать это просто. Для примера я создам два каталога (на этот раз, примеры выполняются в среде ОС UNIX):

```
ops$tkyte@DEV816> create or replace directory  dir1  as  '/trap/';

Directory created.

ops$tkyte@DEV816> create or replace directory "dir2" as  '/tmp/';

Directory created.
```

Пользователь, выполняющий эту операцию, должен обладать привилегией CREATE **ANY DIRECTORY. Я** создал два каталога, чтобы продемонстрировать общую проблему, связанную с регистром символов при работе с объектами-каталогами. При создании первого каталога, **DIR1,** сервер Oracle сохранил объект с именем **в верхнем регистре,** как принято по умолчанию. Во втором примере, **DIR2,** каталог создан с именем, ре-

гистр символов в котором оставлен без изменений. Почему это существенно, будет показано ниже, при использовании объекта типа BFILE.

Предположим, необходимо загрузить данные в столбец типа BLOB или CLOB. Это делается весьма просто, например:

```
ops$tkyte@DEV816> create table demo
  2   (id          int primary key,
  3    theClob    clob
  4   )
  5   /

Table created.

ops$tkyte@DEV816> host echo  'Hello World\!'  > /tmp/test.txt
ops$tkyte@DEV816> declare
  2        l_clob     clob;
  3        l_bfile    bfile;
  4   begin
  5        insert into demo values  (1, empty_dob())
  6        returning theclob into l_clob;
  7
  8     l_bfile    := bfilename('DIR1' , 'test.txt');
  9        dbms_lob.fileopen(l_bfile);
 10
 11        dbms_lob.loadfromfile(l_clob, l_bfile,
 12                                      dbms_lob.getlength(l_bfile));
 13
 14       dbms_lob.fileclose(l_bfile);
 15   end;
 16   /

PL/SQL procedure successfully completed.

ops$tkyte@DEV816> select dbms_lob.getlength(theClob) , theClob from demo
  2   /
    DBMS_LOB.GETLENGTH(THECLOB)  THECLOB

                             13  Hello World!
```

Интересно отметить, что если попытаться выполнить этот пример без изменений в Windows (поменяв, конечно, /tmp/ на каталог, соответствующий этой ОС), результат будет такой:

```
tkyte@TKYTE816> select dbms_lob.getlength(theClob), theClob from demo
  2   /
    DBMS_LOB.GETLENGTH(THECLOB)  THECLOB

                             18   'Hello Horld\!'
```

Длина данных получилась больше из-за того, что командный интерпретатор Windows обрабатывает апострофы и символы маскировки (\) не так, как в ОС UNIX, а признак конца строки здесь длиннее.

Рассмотрим представленный выше код:

- В строках 5 и 6 мы создали строку в таблице, задали столбцу типа **CLOB** значение **EMPTY_CLOB()** и получили это значение — все это одним вызовом. Все большие объекты, за исключением временных, "существуют" в базе данных — мы не можем задать значение переменной типа **LOB,** не сославшись на временный большой объект или большой объект, уже находящийся в базе данных. Пустой объект **EMPTY_CLOB()** — это не Null-значение типа **CLOB;** это непустой указатель на пустую структуру. При этом автоматически получается локатор **LOB,** указывающий на данные в заблокированной строке. Если бы это значение было выбрано без блокирования соответствующей строки, попытки записи в столбец завершились бы неудачно, поскольку перед записью большие объекты должны быть заблокированы (в отличие от других структурированных данных). При вставке строка, безусловно, блокируется. Если бы операция выполнялась с существующей строкой, ее пришлось бы выбирать с конструкцией **FOR UPDATE** для блокирования.
- В строке 8 создается объект типа **BFILE.** Обратите внимание, что имя каталога **DIR1** задано в верхнем регистре — это принципиально важно, как будет показано ниже. Так необходимо делать потому, что функции **BFILENAME** передается **имя** объекта, а не сам объект. Поэтому необходимо убедиться, что имя для объекта задано в том регистре, в котором оно хранится в базе данных Oracle.
- В строке 9 открывается большой объект. Это позволит читать его.
- В строке 11 загружается все содержимое файла операционной системы **/tmp/test.txt** в только что вставленный большой объект, определяемый локатором. Мы используем функцию **DBMS_LOB.GETLENGTH** для передачи подпрограмме **LOADFROMFILE** количества байтов, которые необходимо загрузить из файла **BFILE** (в данном случае грузится все).
- Наконец, в строке 14 открытый объект **BFILE** закрывается; все данные в столбец типа **CLOB** загружены.

Если в представленном выше примере попытаться использовать **dirl** вместо **DIR1,** будет получено следующее сообщение об ошибке:

```
ops$tkyte@DEV816> declare
  2      l_clob      clob;
  3       l_bfile    bfile;
  4  begin
  5       insert into demo values (1, empty_clob())
  6       returning theclob into l_clob;
  7
  8       l_bfile := bfilename('dirl', 'test.txt');
  9       dbms_lob.fileopen(l_bfile);
 10
 11       dbms_lob.loadfromfile(l_clob, l_bfile,
 12                                  dbms_lob.getlength(l_bfile));
 13
 14       dbms_lob.fileclosed(l_bfile);
```

```
 15  end;
 16  /
declare
*
ERROR at line 1:
ORA-22285: non-existent directory or file for FILEOPEN operation
ORA-06512: at "SYS.DBMS_LOB", line 475
ORA-06512: at line 9
```

Причина в том, что каталог **dirl** не существует, а **DIR1** — есть. Если вы предпочитаете использовать в именах каталогов символы разных регистров, необходимо при создании соответствующих объектов использовать идентификаторы в кавычках, как я и сделал для каталога dir2. Это позволит писать код следующего вида:

```
ops$tkyte@DEV816> declare
  2       l_clob     clob;
  3      l_bfile      bfile;
  4  begin
  5       insert into demo values  (2, emptyclob())
  6       returning theclob into l_clob;
  7
  8    l_bfile    := bfilename ('dir2', 'test.txt');
  9      dbms_lob.fileopen(l_bfile);
 10
 11      dbms_lob.loadfromfile(l_clob,  l_bfile,
 12                                    dbms_lob.getlength(l_bfile));
 13
 14      dbms_lob.fileclose(l_bfile);
 15  end;
 16  /

PL/SQL procedure successfully completed.
```

Помимо LOADFROMFILE есть и другие методы, с помощью которых можно наполнять данными столбцы типа LOB в PL/SQL. **LOADFROMFILE** — самое простое решение, если необходимо загрузить весь файл. Если же необходимо обрабатывать содержимое файла в процессе загрузки, можно также применять к объекту **BFILE** функцию **DBMS_LOB.READ** для чтения из него данных. Функция **UTL_RAW.CAST_TO_VARCHAR2** удобна при чтении данных, являющихся текстовыми, а не двоичными. Подробнее о пакете **UTL_RAW** см. в Приложении А. Затем можно использовать вызовы **DBMS_LOB.WRITE** или **WRITEAPPEND** для записи данных в столбец типа **CLOB** или **BLOB.**

Загрузка данных больших объектов с помощью SQLLDR

Теперь разберемся, как загружать данные больших двоичных объектов с помощью утилиты SQLLDR. Для этого существует несколько методов, но мы рассмотрим только два наиболее популярных:

- загрузка данных, находящихся в том же файле, что и основные;

- загрузка данных, хранящихся в отдельных файлах, имена которых указаны в записях файла данных. Такие файлы в SQLLDR принято называть вторичными файлами данных (Secondary Data Files — **SDF**).

Начнем с данных, находящихся в том же файле, что и основные.

Загрузка данных больших объектов из того же файла

В таких больших объектах, как правило, содержатся встроенные символы новой строки и другие специальные символы. Поэтому почти всегда придется использовать один из четырех рассмотренных ранее методов загрузки данных с символами новой строки. Давайте изменим таблицу DEPT так, чтобы столбец COMMENTS стал типа CLOB, а не VARCHAR2:

```
tkyte@TKYTE816> truncate table dept;

Table truncated.

tkyte@TKYTE816> altar table dept drop column comments;

Table altered.

tkyte@TKYTE816> alter table dept add comments clob;

Table altered.
```

Пусть имеется файл данных (demo21.dat) со следующим содержимым:

```
10, Sales,Virginia,01-april-2001,This is the Sales
Office in Virginia|
20,Accounting,Virginia,13/04/2001,This is the Accounting
Office in Virginia|
30,Consulting,Virginia,14/04/2001 12:02:02,This is the Consulting
Office in Virginia|
40,Finance,Virginia,987268297,"This is the Finance
Office in Virginia, it has embedded commas and is
much longer than the other comments field. If you
feel the need to add double quoted text in here like
this: ""You will need to double up those quotes!"" to
preserve them in the string. This field keeps going for up to
1000000 bytes or until we hit the magic end of record marker,
the | followed by a end of line - it is right here ->" |
```

Каждая запись завершается символом конвейера (|), за которым идет маркер конца строки. Как видите, текст для отдела 40 намного длиннее, чем для остальных отделов, содержит много переводов строк, кавычки и запятые. Имея такой файл данных, я могу создать управляющий файл следующего вида:

```
LOAD DATA
INFILE demo21.dat "str X'7C0D0A'"
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ' , ' OPTIONALLY ENCLOSED BY ""
TRAILING NULLCOLS
 (DEPTNO,
```

```
    DNAME         "upper(:dname)",
    LOC           "upper(:loc) ",
    LAST_UPDATED"my_to_date(:last_updated)",
    COMMENTS      char(1000000)
)
```

*Этот пример взят из ОС Windows, в которой строка завершается двумя байтами — отсюда и значение атрибута **STR** в управляющем файле. В ОС UNIX надо использовать значение '7C0A'.*

Для загрузки файла данных я указал тип **CHAR(1000000)** для поля **COMMENTS,** поскольку по умолчанию утилита **SQLLDR** использует для полей данных тип **CHAR(255).** Тип **CHAR(1000000)** позволит **SQLLDR** принимать до 1000000 байт данных. Необходимо указать размер, заведомо превышающий предполагаемую длину текста в поле загружаемого файла. Посмотрим на загруженные данные:

```
tkyte@TKYTE816> select comments from dept;

COMMENTS

This is the Accounting
Office in Virginia

This is the Consulting
Office in Virginia

This is the Finance
Office in Virginia, it has embedded commas and is
much longer then the other comments field.   If you
feel the need to add double quoted text in here like
this: "You will need to double up those quotes!" to
preserve them in the string.   This field keeps going for upto
1,000,000 bytes or until we hit the magic end of record marker,
the |  followed by a end of line - it is right here ->

This is the Sales
Office in Virginia
```

Обратите внимание, что сдвоенные кавычки теперь стали одинарными. Утилита **SQLLDR** удалила лишние кавычки в ходе загрузки.

Загрузка данных больших объектов из внешних файлов

Достаточно часто используется файл данных, содержащий имена файлов, которые необходимо загрузить в большие объекты. Так можно избежать смешивания в одном файле данных больших объектов со структурированными данными. Это дает дополнительную гибкость, поскольку в файле данных не надо применять один из четырех методов обхода проблемы встроенных символов новой строки, которые часто встречаются в больших текстах или двоичных данных. Такие дополнительные файлы данных в **SQLLDR** называются **LOBFILE.**

Утилита **SQLLDR** также поддерживает загрузку файла структурированных данных, ссылающегося на единственный внешний файл данных. Можно указать утилите **SQLLDR,** как выбирать данные больших объектов из этого файла, так что для каждой строки структурированных данных будет загружаться соответствующий его фрагмент. Поскольку такой режим используется очень редко (лично мне он пока ни разу не понадобился), описывать его я не буду. Такие внешние файлы в SQLLDR называются *сложнымивторичнымифайламиданных.*

Файлы **LOBFILE** — относительно простые файлы данных, предназначенные для загрузки больших объектов. Файлы **LOBFILE** от основных файлов данных отличает отсутствие понятия запись, поэтому в них вполне можно использовать символы новой строки. В файлах **LOBFILE** данные могут быть представлены в одном из следующих форматов:

- в виде полей фиксированной длины (например, загрузить байты с 100-го по 1000-ый из файла **LOBFILE);**
- в виде полей с разделителями (поля завершаются чем-то или взяты в "кавычки");
- в виде пар длина-значение, т.е. полей переменной длины.

Чаше всего используется формат с разделителями, причем разделителем является символ **EOF** (конец файла). Обычно имеется каталог с файлами, которые необходимо загрузить в столбцы больших объектов — каждый файл целиком попадает в столбец типа **BLOB.** Для этого используется оператор **LOBFILE** с конструкцией TERMINATED BY EOF.

Итак, допустим, имеется каталог с файлами, которые надо загрузить в базу данных. Необходимо загрузить информацию о владельце файла (OWNER), дату создания файла **(TIMESTAMP),** имя файла **(NAME)** и его данные. Загружать мы будем в следующую таблицу:

```
tkyte@TKYTE816> create table lob_demo
  2   (owner      varchar2(255) ,
  3    timestamp date,
  4    filename  varchar2(255),
  5    text       clob
  6   )
  7   /

Table created.
```

С помощью простой команды **ls -l** в Unix или **dir /q /n** в Windows можно сгенерировать файл данных и загрузить его с помощью такого, например, управляющего файла для ОС Unix:

```
LOAD DATA
INFILE *
REPLACE
INTO TABLE LOB_DEMO
(owner       position(16:24),
 timestamp  position(34:45)  date "Mon DD HH24:MI",
```

```
 filename    position(47:100),
 text LOBFILE(filename)  TERMINATED BY EOF

BEGINDATA
-rw-r--r--  1 tkyte          1785  Sep 27  12:56 demo10.log
-rw-r--r--  1 tkyte          1674  Sep 19  15:57 demo2.log
-rw-r--r--  1 tkyte          1637  Sep 17  14:43 demo3.log
-rw-r--r--  1 tkyte          2385  Sep 17  15:05 demo4.log
-rw-r--r--  1 tkyte          2048  Sep 17  15:32 demo5.log
-rw-r--r--  1 tkyte          1801  Sep 17  15:56 demo6.log
-rw-r--r--  1 tkyte          1753  Sep 17  16:03 demo7.log
-rw-r--r--  1 tkyte          1759  Sep 17  16:41 demo8.log
-rw-r--r--  1 tkyte          1694  Sep 17  16:27 demo8a.log
-rw-r--r--  1 tkyte          1640  Sep 24  16:17 demo9.log
```

В случае Windows, аналогичный управляющий файл будет иметь вид:

```
LOAD DATA
INFILE *
REPLACE
INTO TABLE LOB_DEMO
(owner        position(40:61),
  timestamp   position(1:18)
                    "to_date(:timestamp||'m','mm/dd/yyyy  hhrmiam')",
  filename    position(63:80),
  text LOBFILE(filename)  TERMINATED BY EOF
)
BEGINDATA
04/14/2001   12:36p              1,697 BUILTIN\Administrators demo10.log
04/14/2001   12:42p              1,785 BUILTIN\Administrators demo11.log
04/14/2001   12:47p              2,470 BUILTIN\Administrators demo12.log
04/14/2001   12:56p              2,062 BUILTIN\Administrators demo13.log
04/14/2001   12:58p              2,022 BUILTIN\Administrators demo14.log
04/14/2001   01:38p              2,091 BUILTIN\Administrators demo15.log
04/14/2001   04:29p              2,024 BUiLTIN\Administrators demo16.log
04/14/2001   05:31p              2,005 BUILTIN\Administrators demo17.log
04/14/2001   05:40p              2,005 BUILTIN\Administrators demo18.log
04/14/2001   07:19p              2,003 BUILTIN\Administrators demo19.log
04/14/2001   07:29p              2,011 BUILTIN\Administrators demo20.log
04/15/2001   11:26a              2,047 BUILTIN\Administrators demo21.log
04/14/2001   11:17a              1,612 BUILTIN\Administrators demo4.log
```

Обратите внимание, что мы не загружали данные типа **DATE** в столбец **timestamp** — пришлось использовать SQL-функцию для преобразования формата даты Windows в формат, приемлемый для СУБД. Теперь, если проверить содержимое таблицы **LOB_DEMO** после запуска утилиты SQLLDR, получим:

```
tkyte@TKYTE816> select owner, timestamp, filename, dbms_lob.getlength(text)
  2   from lob_demo;

OWNER                     TIMESTAMP  FILENAME       DBMS_LOB.GETLENGTH(TEXT)

BUILTIN\Administrators    14-APR-01  demo10.log                           1697
BUILTIN\Administrators    14-APR-01  demo11.log                           1785
```

```
BUILTIN\Administrators    14-APR-01  demol2.log                    2470
BOILTIN\Administrators    14-APR-01  demo4.log                     1612
BUILTIN\Administrators    14-APR-01  demol3.log                    2062
BUILTIN\Administrators    14-APR-01  demol4.log                    2022
BUILTIN\Administrators    14-APR-01  demol5.log                    2091
BUILTIN\Administrators    14-APR-01  demol6.log                    2024
BUILTIN\Administrators    14-APR-01  demol7.log                    2005
BUILTIN\Administrators    14-APR-01  demol8.log                    2005
BUILTIN\Administrators    14-APR-01  demol9.log                    2003
BUILTIN\Administrators    14-APR-01  demo20.log                    2011
BUILTIN\Administrators    15-APR-01  demo21.log                    2047

13 rows selected.
```

Этот подход можно применять и для больших двоичных объектов. Загрузить таким способом каталог с изображениями — очень просто.

Загрузка данных больших объектов в объектные столбцы

Теперь, зная, как загружать данные в простую таблицу, специально для этого созданную, можно переходить к загрузке в уже существующую таблицу, включающую столбец сложного объектного типа с атрибутом типа большого объекта. Чаще всего это приходится делать при использовании средств обработки изображений **interMedia** или картриджа Virage Image Cartridge (VIR) в СУБД. Они используют сложный объектный тип **ORDSYS.ORDIMAGE** для столбцов таблиц. Необходимо научиться загружать в них данные с помощью утилиты **SQLLDR.** Для загрузки большого объекта в столбец типа **ORDIMAGE** надо немного разобраться в структуре типа ORDIMAGE. С помощью таблицы, в которую мы будем выполнять загрузку, и простой команды **DESCRIBE,** примененной к этой таблице и используемым типам в SQL*Plus, можно понять, что, имея столбец IMAGE типа **ORDSYS.ORDIMAGE**, данные надо загружать в **IMAGE.SOURCE.LOCALDATA.** Следующий пример можно выполнить, только если установлена и сконфигурирована опция interMedia или картридж Virage Image Cartridge, иначе тип данных **ORDSYS.ORDIMAGE** системе не будет известен:

```
ops$tkyte@ORA8I.WORLD> create table image_load(
  2     id number,
  3    name varchar2(255) ,
  4     image ordsys.ordimage
  5  )
  6  /
Table created.
ops$tkyte@ORA8I.WORLD> desc image_load
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ID                                                 NUMBER
 NAME                                               VARCHAR2(255)
 IMAGE                                              ORDSYS.ORDIMAGE
ops$tkyte@ORA8I.WORLD> desc ordsys.ordimage
```

```
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------------------
 SOURCE                                             ORDSOURCE
 HEIGHT                                             NUMBER(38)
 WIDTH                                              NUMBER(38)
 CONTENTLENGTH                                      NUMBER(38)

ops$tkyte@ORA8I.WORLD> desc ordsys.ordsourca
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------------------
 LOCALDATA                                          BLOB
 SRCTYPE                                            VARCHAR2 (4000)
 SRCLOCATION                                        VARCHAR2(4000)
```

Управляющих файл для загрузки может выглядеть так:

```
LOAD DATA
INFILE *
INTO TABLE image_load
REPLACE
FIELDS TERMINATED BY ','
(ID,
 NAME,
 file_name FILLER,
 IMAGE column object
 (
   SOURCE column object
   (
     LOCALDATA LOBFILE (file_name) TERMINATED BY EOF
                NULLIF file name = 'HONE'


BEGINDATA
1,icons,icons.gif
```

В нем я добавил две новые конструкции:

- **COLUMN OBJECT.** Этот тип поля сообщает **SQLLDR,** что задано не имя столбца, а часть имени. Она не сопоставляется с полем файла данных, а используется для построения корректных ссылок на столбцы объекта при загрузке. В представленном примере есть два вложенных признака объекта. Поэтому будет использоваться имя столбца **IMAGE.SOURCE.LOCALDATA,** что и требуется. Обратите внимание, что мы не загружаем никакие другие атрибуты этих объектных типов, например **IMAGE.HEIGHT, IMAGE.CONTENTLENGTH, IMAGE.SOURCE.SRCTYPE.** Ниже мы увидим, как задать их значения.
- **NULLIF FILE_NAME = 'NONE'.** Эта конструкция требует от **SQLLDR** загружать Null в объектный столбец, если поле **FILE_NAME** содержит слово **NONE.**

После загрузки в столбцы interMedia-типов необходимо выполнить завершающую обработку загруженных данных с помощью PL/SQL, чтобы компонент **interMedia** мог с

ними работать. Например, в нашем случае, вероятно, понадобится выполнить следующую обработку для корректной установки соответствующих свойств изображений:

```
begin
   for c in (select * from image_load) loop
     c.image.setproperties;
   end loop;
end;
/
```

SETPROPERTIES — это метод объекта, предоставляемый типом ORDSYS.ORDIMAGE, который обрабатывает изображение и изменяет соответственно остальные атрибуты объекта. Подробнее обработка изображений в **interMedia** описана в главе 17.

Загрузка массивов переменной длины и вложенных таблиц с помощью SQLLDR

Теперь давайте рассмотрим, как с помощью утилиты SQLLDR загружать массивы переменной длины и вложенные таблицы. Массивы переменной длины и вложенные таблицы (с этого момента мы будем называть их массивами) будут задаваться во входном файле следующим образом.

- В файл данных включается дополнительное поле. Оно будет содержать количество элементов массива, которые должны быть в файле данных. Поле не загружается, а используется утилитой SQLLDR для определения количества элементов в загружаемом массиве.
- Затем указывается поле или набор полей, задающих элементы массива.

Итак, предполагается, что массивы переменой длины в файле данных снабжены дополнительным полем количества элементов, за которым следуют сами элементы. Можно также загружать массивы данных с одинаковым, фиксированным количеством элементов в каждой записи (например, в каждой входной записи — пять элементов массива). Мы рассмотрим оба метода на базе представленного ниже типа:

```
tkyte@TKYTE816> create type myArrayType
  2  as varray(10) of number(12,2)
  3  /
Type created.
tkyte@TKYTE816> create table t
  2   (x int primary key, y my_Array_Type)
  3  /

Table created.
```

Это схема, в которую будут загружаться данные. А вот пример управляющего файла сданными, который можно использовать для загрузки. Он демонстрирует загрузку массивов с переменным количеством элементов. Каждая входная запись будет иметь формат:

- Значение x.

- Количество элементов в y.
- Поля, содержащие отдельные элементы y.

```
LOAD DATA
INFILE *
INTO TABLE t
replace
fields terminated by ","
(
  x,
  y_cnt              FILLER,
  y                    varray count (y_cnt)
  (
    y
  )
)
BEGINDATA
1,2,3,4
2,10,1,2,3,4,5,6,7,8,9,10
3,5,5,4,3,2,1
```

Обратите внимание, что использовано ключевое слово **FILLER,** позволяющее присвоить значение поля переменной **Y_CNT,** не загружая его. Также использована конструкция **VARRAY COUNT (Y_CNT),** сообщающая **SQLLDR,** что y имеет тип **VARRAY.** Если бы поле Y представляло собой вложенную таблицу, пришлось бы использовать конструкцию **NESTED TABLE COUNT(Y_CNT).** Учтите также, что это — ключевые слова **SQLLDR,** а не функции **SQL,** поэтому для связываемых переменных не используются кавычки или двоеточия, как при ссылке на них в функции SQL.

Ключевым элементом является конструкция **COUNT.** Она позволяет получить значение загружаемого массива. Например, получив следующую строку данных:

```
1,2,3,4
```

мы разбираем ее так:

- 1 сопоставляется с x — первичным ключом;
- 2 сопоставляется с **y_cnt** — количеством элементов массива;
- 3,4 сопоставляется с y — элементами массива.

После запуска **SQLLDR** получим:

```
tkyte@TKYTE816> select * from t;

         X Y
---------- ----------------------------------------
         1 MYARRAYTYPE(3, 4)
         2 MYARRAYTYPE(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
         3 MYARRAYTYPE(5, 4, 3, 2, 1)
```

Именно это и требовалось. Теперь, допустим, необходимо загрузить входной файл с фиксированным количеством элементов в каждом массиве. Например, имеется иденти-

фикатор и пять значений, связанных с этим идентификатором. Можно использовать следующий тип и таблицу:

```
tkyte@TKYTE816> create or replace type myTableType
  2  as table of number(12,2)
  3  /

Type created.

tkyte@TKYTE816> create table t
  2   (x int primary key, y myTableType)
  3  nested table y store as y_tab
  4  /

Table created.
```

Управляющий файл будет выглядеть следующим образом: (обратите внимание на использование CONSTANT 5 в конструкции count для вложенной таблицы; она указывает утилите SQLLDR, сколько элементов будет в каждой записи)

```
LOAD DATA
INFILE *
INTO TABLE t
replace
fields terminated by ","
(
  x,
  y                       nested table count  (CONSTANT 5)
   (
     y
   )
)

BEGINDATA
1,100,200,300,400,500
2,123,243,542,123,432
3,432,232,542,765,543
```

После запуска SQLLDR получим:

```
tkyte@TKYTE816> select * from t;

      X Y

      1 MYTABLETYPE(100, 200, 300, 400, 500)
      2 MYTABLETYPE(123, 243, 542, 123, 432)
      3 MYTABLETYPE(432, 232, 542, 765, 543)
```

Как видите, данные загружены во вложенную таблицу. Отвлекаясь ненадолго от темы SQLLDR, хочу обратить ваше внимание на то, что при загрузке можно обнаружить одно из свойств вложенных таблиц. Я случайно обнаружил, что при повторной загрузке тех же данных в таблицу с помощью SQLLDR получается немного другой результат:

```
tkyte@TKYTE816> host sqlldr userid=tkyte/tkyte control=demo24.ctl
SQLLDR: Release 8.1.6.0.0 - Production on Sun Apr 15 12:06:56 2001
```

```
(c) Copyright 1999 Oracle Corporation.  All rights reserved.

Commit point reached - logical record count 3

tkyte@TKYTE816> select * from t;

      X Y
------- ----------------------------------------
      1 MYTABLETYPE(200, 300, 400, 500, 100)
      2 MYTABLETYPE(123, 243, 542, 123, 432)
      3 MYTABLETYPE(432, 232, 542, 765, 543)
```

Обратите внимание, что теперь число 100 — **последнее** в первой вложенной таблице. Это побочный эффект повторного использования пространства в таблице при повторной загрузке. Он может воспроизводиться на других системах или с другим размером блока (а может и — нет, как произошло у меня). Во вложенных таблицах порядок строк не сохраняется, так что не удивляйтесь, если данные во вложенной таблице оказываются не в том порядке, как вы их загружали!

Вызов утилиты SQLLDR из хранимой процедуры

Если коротко — этого сделать нельзя. **SQLLDR** — не набор функций, его нельзя вызвать. **SQLLDR** — это утилита командной строки. Конечно, можно написать внешнюю процедуру на языке Java или C, которая запускает **SQLLDR** (см. в главе 19 пример хранимой процедуры для выполнения команды операционной системы), но это не то же самое, что "вызвать" **SQLLDR.** Загрузка при этом будет происходить в другом сеансе, вне вашей транзакции. Кроме того, придется анализировать полученный в результате журнальный файл, чтобы определить, была ли загрузка успешной и насколько успешной (сколько строк было загружено, прежде чем загрузка была прекращена из-за возникновения ошибок). Я не рекомендую вызвать **SQLLDR** из хранимой процедуры.

Итак, что же делать, если необходимо загрузить данные в хранимой процедуре? Возможны следующие варианты.

- Написать мини-утилиту **SQLLDR** на языке PL/SQL. При этом можно использовать либо переменные типа **BFILE** для чтения двоичных данных, либо пакет **UTL_FILE** для чтения, анализа и загрузки текстовых данных. Этот подход будет продемонстрирован далее.
- Написать мини- утилиту **SQLLDR** на языке Java. Это не сложнее, чем создание загрузчика на PL/SQL, поскольку можно использовать многие существующие компоненты Java.
- Написать **SQLLDR** в виде функции на C, и вызывать его как внешнюю процедуру-

Я представил эти варианты в порядке повышения сложности и производительности. Обычно чем сложнее решение, тем выше его производительность. В нашем случае реализации на языках PL/SQL и Java сравнимы по производительности, а реализация на C

будет работать быстрее (но окажется менее переносимой и более сложной в создании и установке). Я люблю простоту и переносимость, поэтому продемонстрирую идею реализации на примере PL/SQL. Удивительно, как просто, оказывается, написать собственный мини-SQLLDR. Например:

```
ops$tkyte@DEV816>  create  table  badlog(ernn varchar2(4000),
  2                              data varchar2(4000));

Table  created.
```

Начнем с таблицы для записей, которые не удалось загрузить. Затем создадим функцию загрузки:

```
ops$tkyte@DEV816> create or replace
  2   function   load_data(p_table      in  varchar2,
  3                        p_cnames     in  varchar2,
  4                        p_dir        in varchar2,
  5                        p_filename   in  varchar2,
  6                        p_delimiter  in  varchar2  default  '|')
  7   return number
```

Она принимает имя таблицы, в которую должна выполняться загрузка, список имен столбцов в том порядке, в каком они заданы во входном файле, каталог и имя загружаемого файла, а также разделитель данных во входном файле. Функция возвращает количество успешно загруженных записей. Далее идут локальные переменные этой небольшой функции:

```
  8   is
  9     l_input       utl_file.file_type;
 10     l_theCursor    integer default  dbms_sql.open_cursor;
 11     l_buffer      varchar2(4000);
 12     l_lastLine    varchar2(4000);
 13     l_status       integer;
 14     l_colCnt       number default  0;
 15     l_cnt         number default  0;
 16     l_sep          char(1)  default NULL;
 17     l_errmsg      varchar2(4000);
 18   begin
```

Затем мы открываем входной файл. Предполагается загружать простые данные через разделитель, причем длина строк не должна превышать 4000 байт. Это ограничение можно расширить до 32 Кбайт (максимальный размер, поддерживаемый пакетом UTL_FILE). Для работы с записями большего размера придется использовать тип данных **BFILE** и средства пакета **DBMS_LOB**:

```
 19            /*
 20              * Открываем файл, из которого считываются данные.
 21              * Предполагается, что он состоит из простых записей через
                 * разделитель.
 22             */
 23       l_input  := utl_file.fopen(p_dir, p_filename,  'r',  4000);
```

Теперь создадим оператор **INSERT** вида **INSERT INTO TABLE (столбцы) VALUES (связываемые переменные).** Количество вставляемых столбцов определяем путем подсчета запятых. Для этого берем длину текущего списка столбцов, вычитаем длину той же строки после удаления запятых и прибавляем 1 (это общий способ подсчета количества определенных символов в строке):

```
24
25      l_buffer := 'insert into ' || p_table ||
26           '('||            p_cnames || ') values (';
27           /*
28            * Определяем количество запятых путем вычитания
29            * из текущей длины списка имен столбцов
30            * длины той же строки с удаленными запятыми
31            * и прибавления 1.
32            */
33      l_colCnt := length(p_cnames)-
34                      length(replace(p_cnames,',',''))+1;
35
36      for i in 1 .. l_colCnt
37       loop
38      l_buffer       := l_buffer || l_sep || ':b'||i;
39        l_sep    := ',';
40       end loop;
41      l_buffer := l_buffer || ' ) ' ;
42
43           /*
44            * Получили строку вида:
45            * insert into T (c1,c2,...) values (:b1, :b2, ...)
46            */
```

Теперь, сформировав строку для оператора **INSERT,** разберем ее:

```
47      dbms_sql.parse(l_theCursor, l_buffer, dbms_sql.native);
```

а затем прочитаем последовательно строки входного файла, разбивая их на столбцы:

```
48
49       loop
50          /*
51           * Читаем данные, пока они есть, затем завершаем работу.
52           */
53           begin
54               utl_file.get_line(l_input, l_lastLine);
55           exception
56                when NO_DATA_FOUND then
57                     exit;
58          end;
59           /*
60            * Анализировать строку удобнее, если она завершается
61            * разделителем.
62            */
63           l_buffer := l_lastLine || p_delimiter;
64
```

```
65
66              for i in 1 .. l_colCnt
67              loop
68                  dbms_sql.bind_variable(l_theCursor,  ':b'||i,
69                                         substr(l_buffer, 1,
70                                             instr(l_buffer,p_delimiter)-1)) ;
71          l_buffer          := substr(l_buffer,
72                   instr(l_buffer,delimiter)+l)                    ;
73              end loop;
74
75              /*
76               * Выполняем оператор insert.  В случае ошибки
77               * помещаем строку данных в таблицу "плохих" строк.
78               */
79              begin
80                  l_status := dbms_sql. execute(l_theCursor);
81                  l_cnt := l_cnt + 1;
82              exception
83                  when others then
84                       l_errmsg  := sqlerrm;
85                       insert into badlog  (errm,  data)
86                       values  (l_errmsg,  l_lastLine);
87             end;
88        end loop;
```

После загрузки всех записей, которые можно было загрузить, и помещения всех остальных в таблицу "плохих" строк, завершаем работу и возвращаем результат:

```
89
90        /*
91         * закрыть курсор,  файл и зафиксировать записи
92         */
93        dbms_sql.close_cursor(l_theCursor);
94        utl_file.fclose(l_input);
95        commit;
96
97        return  1 cnt;
98   exception
99              when others then
100             dbms_sql.close_cursor(l_theCursor);
101                       if  (utl_file.is_open(  l_input  ))  then
102                                utl_file.fclose(l_input);
103                      end if;
104                       RAISE;
105  end load_data;
106  /
```

```
Function  created.
```

Эту функцию можно использовать следующим образом:

```
ops$tkyte@DEV816> create  table  tl  (x  int,  y  int,  z  int);

Table  created.
```

```
ops$tkyte@DEV816> host echo 1,2,3 > /tmp/t1.dat
ops$tkyte@DEV816> host echo 4,5,6 >> /tmp/t1.dat
ops$tkyte@DEV816> host echo 7,8,9 >> /tmp/t1.dat
ops$tkyte@DEV816> host echo 7,NotANumber,9 >> /tnp/t1.dat

ops$tkyte@DEV816> begin
  2      dbms_output.put_line<
  3           load_data('T1',
  4                     'x,y,z',
  5                     'c:\temp',
  6                     't1.dat',
  7        ',')        ||        ' rows loaded') ;
  8  end;
  9  /
3 rows loaded

PL/SQL procedure successfully completed.

ops$tkyte@DEV816> SELECT *
  2     FROM BADLOG;

ERRM                        DATA
--------------------------- ----------------------------
ORA-01722: invalid number 7,NotANumber,9

ops$tkyte@DEV816> select * from t1;

        X      Y      Z

        1      2      3
        4      5      6
        7      8      9
```

Конечно, это решение уступает по гибкости **SQLLDR,** поскольку нет способа задать максимально допустимое количество ошибок или указать, куда помещать "плохие" записи, нельзя задать символ, в который могут быть взяты значения полей, и т.д., но понятно, насколько легко добавить эти возможности. Например, чтобы добавить необязательный символ "кавычек" вокруг значений полей, достаточно добавить параметр **P_ENCLOSED_BY** и заменить вызов **DBMS_SQL.BIND** следующим:

```
loop
     dbms_sql.bind_variable(l_theCursor, ':b'||i,
     trim(nvl(p_enclosed_by,chr(0))FROM
                         substr(l_buffer, 1,
                              instr(l_buffer,p_delimiter)-1));
     l_buffer := substr(l_buffer,
                        instr(l_buffer,p_delimiter)+1);
end loop;
```

сымитировав опцию **OPTIONALLY** ENCLOSED BY утилиты SQLLDR. Представленная выше функция вполне подходит для загрузки небольших объемов данных, однако, как будет показано в главе 16, при необходимости масштабирования в нее придется добавить подпрограмму для обработки массивов данных. В этой главе можно найти примеры использования массивов для множественной вставки данных.

На Web-сайте издательства Wrox можно найти еще один мини-утилиту **SQLLDR** на языке PL/SQL. Она создана специально для загрузки файлов в формате **dBASE**. Для чтения данных используются объекты типа **BFILE,** поскольку файлы формата **dBASE** содержат одно-, двух- **и** четырехбайтовые целые числа, которые не позволяет обрабатывать пакет **UTL_FILE**. Загрузчик может либо описывать содержимое dBASE-файла, либо загружать его в базу данных.

Проблемы

Давайте рассмотрим ряд особенностей, которые необходимо учитывать при использовании утилиты **SQLLDR.**

Нельзя выбрать сегмент отката

Часто при загрузке с помощью **SQLLDR** используется опция **REPLACE.** В результате перед загрузкой данных просто выполняется оператор **DELETE.** При этом может генерироваться огромный объем данных отката. Хотелось бы помещать их в конкретный, специально созданный большой сегмент отката. Однако утилита **SQLLDR** не предлагает средств привязки к сегменту отката. Необходимо обеспечить достаточный размер всех сегментов отката, чтобы в них поместились данные отката для оператора **DELETE, или** просто использовать опцию **TRUNCATE.** Поскольку при выполнении оператора **INSERT** данных отката генерируется мало, а утилита **SQLLDR** достаточно часто выполняет фиксацию, такая проблема возникает только при использовании опции **REPLACE.**

TRUNCATE работает по-другому

Опция **TRUNCATE** в **SQLLDR** может работать не так, как оператор TRUNCATE в **SQL*Plus** или других инструментальных средствах., Исходя из предположения, что в таблицу будет загружен примерно такой же объем данных, утилита **SQLLDR** использует расширенный вариант оператора **TRUNCATE.** Она выполняет:

```
truncate table t reuse storage
```

Опция REUSE STORAGE не освобождает выделенные экстенты, она лишь помечает их как "свободное пространство". Если такой результат нежелателен, необходимо явно выполнять оператор **TRUNCATE** для таблицы перед вызовом **SQLLDR.**

Стандартным типом поля в SQLLDR является CHAR(255)

Стандартная длина поля — 255 символов. Если загружаемое поле длиннее, выдается сообщение об ошибке:

```
Record N: Rejected - Error on table T, column C.
Field in data file exceeds maximum length
```

Это не означает, что данные не поместятся в столбец базы данных, просто SQLLDR ожидает не более 255 байт данных, а получает больше. Решение: задать тип поля CHAR(N) в управляющем файле, где N — достаточно большое значение, превышающее размер самого большого поля во входном файле.

Опции командной строки переопределяют установки в командном файле

Многие опции утилиты SQLLDR можно задавать как в управляющем файле, так и в командной строке. Например, можно использовать конструкцию INFILE в управляющем файле, а можно просто вызывать **SQLLDR ... DATA=FILENAME**. Опции командной строки переопределяют соответствующие установки командного файла. Нельзя гарантировать, что будут использоваться именно установки управляющего файла, поскольку при вызове SQLLDR пользователь их может переопределить.

Резюме

В этой главе рассмотрены стандартные задачи: загрузка файлов с разделителями; загрузка файлов с записями фиксированной длины; загрузка файлов изображений из каталога; преобразование данных с помощью функций в ходе загрузки; способы выгрузки данных и т.д. Мы не рассматривали подробно множественную загрузку данных в режиме непосредственной загрузки, а лишь кратко описали ее особенности. В главе мы хотели ответить на вопросы, наиболее часто возникающие при использовании SQLLDR и интересующие максимально широкую аудиторию.

10

Стратегии и средства настройки

Мне очень хорошо знакомы проблемы настройки. Я потратил уйму времени на настройку систем, в особенности тех, которые не проектировал и не реализовывал. Это создает определенные трудности: надо подумать не только о том, где искать проблемы, но и о том, где не искать. Настройка при этом становится чрезвычайно сложной и ее приходится выполнять в крайне жестких условиях. Никто не занимается настройкой, пока все хорошо, — о ней начинают думать, когда система разваливается.

В этой главе описан подход и средства, используемые мною при настройке. Я стараюсь делать все так, чтобы ничего никогда не приходилось настраивать; по крайней мере — систему после ввода в эксплуатацию. Настройка — часть процесса разработки, начинающаяся еще до того, как написана первая строка кода, и заканчивающаяся за день до внедрения; настройкой нельзя заниматься после внедрения. К сожалению, большинство проблем настройки, к решению которых меня привлекают, связано с настройкой уже созданных производственных систем. Это означает, что настройка ведется во враждебной среде (недовольных пользователей) и в условиях множества нежелательных ограничений (никто не хочет останавливать производственную систему, чтобы изменить в ней что-то). Лучше всего настраивать задолго до этого момента.

В частности, в этой главе рассматривается:

- использование связываемых переменных и их влияние на производительность;
- выполнение трассировки приложений с помощью установок **SQL_TRACE** и **TIMED_STATISTICS**, а интерпретация результатов с помощью утилиты **TKPROF**;
- установка и использование пакета **Statspack** для настройки экземпляра;
- использование представлений **V$**, к которым я регулярно обращаюсь.

Определение проблемы

Настройка приложения может оказаться действительно интересным занятием. Достаточно сложно понять, где приложение работает не так, исправить же ситуацию еще сложнее. В некоторых случаях необходимо менять общую архитектуру. В главе 1 я описывал систему, выполнявшую продолжительные хранимые процедуры на многопоточном сервере Oracle. В этой системе пришлось полностью пересмотреть архитектуру. В других случаях достаточно было просто найти наименее производительные SQL-запросы и настроить их.

Не вся настройка связана с базой данных. Я помню один особенно сложный случай настройки, когда клиент использовал коммерческое приложение по учету рабочего времени на базе СУБД Oracle. Это приложение уже было установлено в нескольких местах и везде успешно работало. Однако здесь, где пользователей было наибольшее количество, оно полностью разваливалось. Большую часть времени оно работало прекрасно, как и ожидалось. При пиковой нагрузке, например при пересменке или в обеденный перерыв, система периодически зависала. Причем по непонятным причинам. По всем признакам мне было понятно, что проблема связана с блокированием/конфликтами доступа — это было очевидно. А вот понять причины блокирования и конфликтов оказалось непросто. После двух дней поиска в базе данных, просмотра всех представлений V$, изучения кода приложения и признаков проблемы я попросил показать приложение в действии. На концептуальном уровне я понимал, что оно делает, но не знал, как именно делает. Меня привели на склад, где приложение использовали пользователи (фабричные рабочие). Я увидел, как именно они его используют. Они выстраивались в очереди у терминалов и проводили карточкой со штрих-кодом по считывателю, чтобы зафиксировать время прихода. Следующий рабочий в очереди нажимал клавишу Enter, проводил карточкой по считывателю и шел дальше. Неожиданно, пока я там стоял, приложение зависло. Ни одна карточка не считывалась. Потом кто-то подошел к свободному терминалу, нажал клавишу Enter и провел карточкой по считывателю — система заработала! При более детальном анализе приложения стало понятно, что происходит. Оказалось, проблема вообще не была связана с базой данных — это была проблема взаимодействия человека и компьютера. Причиной зависания был интерфейс приложения. Выяснилось, что система создавалась для обработки транзакций следующего вида:

- считывается карточка, при этом блокируется строка в таблице и вставляется строка в другую таблицу;
- на экран сообщение выдается о том, что строка вставлена, при этом надо нажать клавишу Enter;
- пользователь нажимает Enter, и приложение фиксирует транзакцию (до этого момента транзакция еще не зафиксирована);
- следующий пользователь проводит карточкой по считывателю.

На самом деле рабочие делали так:

- проводили карточкой по считывателю и уходили;
- следующий человек в очереди нажимал Enter вместо предыдущего, фиксируя его проход, проводил карточкой по считывателю и уходил.

После того как считывалась карточка последнего человека в очереди, транзакция оставалась открытой, что блокировало ресурсы. Фоновый процесс, срабатывающий раз в несколько минут, блокировал некоторые ресурсы, а затем пытался заблокировать тот же ресурс, что и "открытая" транзакция. Фоновый процесс останавливался, предварительно заблокировав некоторые ресурсы, необходимые интерактивному приложению. В результате все терминалы блокировались и система "зависала", как и было отмечено, пока кто-нибудь, проходя мимо терминала, не замечал сообщение с просьбой нажать клавишу Enter для продолжения работы и не нажимал эту клавишу. После этого все опять работало отлично. Простое изменение в пользовательском интерфейсе — немедленная фиксация транзакции при считывании карточки — решило проблему.

В другом случае, в одной из моих последних миссий по настройке, пришлось иметь дело с очень большим приложением. В его разработке, реализации и внедрении участвовали сотни людей. Ситуация достигла кризисной точки (так всегда бывает, когда приходится заниматься настройкой). Проблема: "приложение Oracle работает медленно". После беглого изучения приложения, среды и базы данных причина проблемы оставалась неочевидной. При более детальном изучении приложения несколько уязвимых точек системы стали вырисовываться. Как оказалось, проблема была вовсе не в приложении Oracle, а в интерфейсе к существующей системе. Новое приложение использовало существующую систему так, как никто не предполагал. Существующая система не справилась с дополнительной нагрузкой. Новое приложение добило старое. Выяснить это оказалось непросто, поскольку код не был для этого подготовлен ("подготовка" в данном случае означала просто наличие "отладочных" сообщений или журнала приложения с отметкой времени выполнения каждой операции, чтобы можно было увидеть, что происходит). Нам пришлось добавить много подобного "инструментария" постфактум, чтобы выяснить причину замедления (и даже периодической остановки) работы системы.

Мораль этих историй в том, что настройка — дело непростое, и решения здесь не всегда интуитивно понятны. Две проблемы настройки не решаются абсолютно одинаково; проблемы не всегда связаны с базой данных, они не всегда находятся в приложении и далеко не всегда вызваны несоответствием архитектуры. Поиск проблем, особенно когда непонятно, как должно использоваться приложение или как оно работает, иногда ведется "на удачу". Вот почему некоторые относят настройку к области "черной магии". Очень сложно объяснить кому-нибудь, как настраивать систему, если настраивать приходится постфактум. Настройка готового приложения требует опыта расследования, как у хорошего детектива — вы открываете тайну. Для этого требуется соответствующее сочетание технических знаний и опыта работы с людьми (никто не хочет, чтобы на него потом показывали пальцем, и этот аспект надо учитывать). Нет ни простых, ни сложных готовых путей настройки постфактум. Однако я могу рассказать вам, как настраивать по ходу разработки — именно такой стратегии я рекомендую придерживаться. С моей точки зрения, настройкой системы надо заниматься при проектировании. Производительность должна обеспечиваться на всех уровнях системы. Настройка системы постфактум — это фактически не настройка, а переписывание кода и изменение архитектуры.

Мой подход

В этом разделе я хочу представить ряд общих принципов настройки. Если и есть что-нибудь в Oracle, относящееся к "шаманству", так это настройка производительности. Все, что вы не понимаете, кажется чудом. Настройку баз данных многие как раз и не понимают. Мне же искусство настройки кажется вполне осваиваемым. Я уверен, что существует три уровня настройки, о которых надо знать и которые необходимо проходить последовательно.

- **Настройка приложения, часть 1.** Настройка изолированного приложения. Обеспечение его максимально быстрой работы в однопользовательском режиме.
- **Настройка приложения, часть 2.** Настройка приложения в многопользовательском режиме. Обеспечение поддержки как можно большего количества одновременно работающих пользователей.
- **Настройка экземпляра/сервера.**

Настройка приложения, изолированно и в многопользовательском режиме, требует **более 90 процентов** всех усилий по настройке. Да, прежде чем обращаться к администратору базы данных мы, разработчики, делаем уже 90 процентов работы. Именно поэтому, я думаю, многие и не понимают, в чем состоит настройка базы данных. Они постоянно просят меня настроить им базу данных и вообще не трогают свои приложения! За исключением экзотических случаев, это физически невозможно. Все ищут то, что я называю магическим параметром инициализации **fast=true.** Этот магический параметр заставил бы их базу данных работать быстрее. Чем раньше вы смиритесь с тем, что такого параметра нет, тем лучше для вас. Крайне маловероятно, что можно заставить запросы выполняться существенно быстрее простой установкой параметра в файле инициализации. Может потребоваться реорганизация данных, причем я имею в виду не распределение файлов данных по дискам для ускорения ввода/вывода — я говорю об изменении физического порядка столбцов в таблицах, об изменении количества и содержимого таблиц. Это означает переделку приложения.

На уровне базы данных тоже есть что настраивать, но опыт показывает, что большая часть проблем настройки решается на уровне приложений. Наиболее вероятно, что именно автор приложения сможет заменить запрос, для получения ответа на который выполняется 1000000 логических операций ввода/вывода, другим запросом или найти альтернативный способ получения той же информации. Если проблемы связаны с архитектурой приложения, только его автор сможет их исправить или обойти.

Настройка - непрерывный процесс

Если вы поняли, что основные возможности для настройки надо искать на уровне приложения, следующий шаг — уяснить, что настройкой надо заниматься непрерывно. У этого процесса нет начала и конца. Настройка производительности является частью этапа проектирования, она выполняется на этапе разработки, в ходе тестирования, при внедрении системы и затем при ее эксплуатации.

Проектирование с учетом производительности

Системы баз данных нельзя создавать по принципу "сегодня создаем, завтра — настраиваем". Слишком много решений, непосредственно влияющих на производительность и, что еще важнее, масштабируемость системы, будет принята на этапе "создания". Помните, ранее я уже писал, что для СУБД Oracle намного проще создать немасштабируемую систему, чем хорошо масштабируемую. Создать немасштабируемое приложение легко — это может сделать каждый. А вот для создания быстро работающего **и** масштабируемого — придется потрудиться. Оно должно с самого начала проектироваться соответствующим образом.

История из жизни: группа проектировщиков разработала приложение. Оно являлось расширением существующей системы. Приложение включало ряд собственных таблиц и использовало несколько существующих. Новое приложение было внедрено, и сразу же существующая система стала неработоспособной — вся система в целом стала работать слишком медленно. Оказалось, что разработчики нового приложения решили, что разберутся, какие индексы нужны, "на месте". За исключением созданных автоматически для поддержки первичных ключей, никаких индексов не было. Практически каждый запрос приводил к многочисленным полным просмотрам таблиц. Разработчики спрашивали, как это быстро исправить. Напрашивался лишь один ответ: "Удалите ваши таблицы и выкиньте приложение". Проектировщикам казалось, что настройкой можно заняться по завершении разработки, т.е. сначала создаем работающее приложение, а затем заставляем его работать быстрее. Не стоит и говорить, что проект закончился полным провалом. Разработанная структура базы данных не позволяла эффективно отвечать на выполняемые запросы. Пришлось вернуться к началу и все создавать заново.

Для того чтобы проиллюстрировать свой подход, расскажу небольшую историю. Там, где я работал, использовалась простая внутренняя система под названием **'phone'.** Можно было с помощью **telnet** подключиться к любому почтовому серверу (тогда почта была только текстовой) и в командной строке набрать **phone <искомая строка>.** В ответ выдавались данные следующего вида:

```
$ phone tkyte
TKYTE    Kyte, Tom              703/555 4567  Managing Technologies
RESTON:
```

Примерно в 1995/1996 году, когда активно начал использоваться Web, наша группа создала небольшую Web-систему, загружавшую данные о телефонах в таблицу и позволявшую пользователям вести по этой таблице поиск. Теперь, после помещения данных в СУБД и добавления небольшого графического интерфейса, система стала в компании стандартным средством поиска информации о сотрудниках. Со временем мы стали добавлять в нее все больше данных и дополнительные поля. Она становилась все более популярной. В определенный момент мы решили добавить намного больше полей и пересоздать систему, существенно расширив ее возможности.

Нашей целью на этой стадии было сразу вести разработку с учетом производительности. Мы знали, что создаваемая система будет ограничивающим фактором производительности сервера в целом. Хотя и небольшая по размеру кода, эта система должна была обеспечивать основной объем информации, получаемой с сервера. Первое, что мы сделали, с учетом наших знаний и предположений о будущем использовании этой про-

стенькой системы, — спроектировали таблицы, в которых содержатся ее данные. Мы специально проектировали эти таблицы с учетом дальнейшего использования. Речь шла о небольшом хранилище данных только для чтения, в котором пользователи осуществляют поиск, причем поиск должен был выполняться быстро. Популярность системы росла с каждым днем, и система угрожала захватить все ресурсы сервера. Вся информация хранилась в одной 67-столбцовой таблице с 75000 строк, в которой необходимо было выполнять поиск строк по разным полям. Так что, если ввести строку **ABC,** то ее поиск выполняется в поле адреса электронной почты, имени и т.д. Что еще хуже, можно было вводить шаблоны типа **%ABC%** и искать без учета регистра символов.

Ни в одной существующей СУБД нет индекса, который поддерживал бы подобный поиск, поэтому пришлось сделать собственный. Каждую ночь, при получении обновленных данных из системы учета персонала (делалось полное обновление данных в таблице), мы выполняли следующий оператор:

```
CREATE TABLE FAST_EMPS
PCTFREE 0
CACHE
AS
SELECT  upper(last_name)||'/'||upper(first_name)||'/'  ||  ' /' ||
                      substr(phone, length(phone)-4) SEARCH_STRING,
        rowid row_id
  FROM EMPLOYEES
/
```

после завершения обновления. Фактически мы строили максимально плотную и компактную таблицу **(pctfree 0)** и рекомендовали держать ее в буферном кэше.

```
select *
  from employees
 where rowid in (select row_id
                   from fast_emp
                  where search_string like :bv
                    and rownum <= 500)
```

Этот запрос всегда выполняет **полный просмотр** таблицы FAST_EMP, но мы этого и добивались. С учетом типа выполняемых запросов, это был единственно возможный выбор. Нет никакой схемы индексации, поддерживающей все запросы требуемого вида. Нашей целью было минимизировать объем просматриваемых данных, ограничить объем данных, получаемых в ответ на запросы, и сделать поиск максимально быстрым. Представленный выше подход позволяет достичь всех трех целей. Таблица FAST_EMP обычно целиком будет помещаться в буферном кэше. Она — маленькая (размер ее составляет менее 8 процентов размера исходной таблицы) и просматривается очень быстро. Поиск без учета регистра символов в ней обеспечен уже при создании, один раз (а не при каждом запросе), путем хранения данных в верхнем регистре. Количество соответствующих запросу записей не будет превосходить 500 (если поиск дает больше результатов, надо уточнить критерий: 500 результатов все равно никто **никогда** не просматривает). Фактически эта таблица используется во многом аналогично индексу, поскольку хранит идентификаторы строк в основной таблице. Для поддержки поиска в этой системе вообще не использовались индексы — только две таблицы.

Это прекрасный пример приложения с очень специфическими требованиями к проекту, а также пример того, как при учете этих требований в самом начале можно получить оптимальную производительность.

Пробуйте разные подходы

Очень важно экспериментировать, пробуя различные реализации. Теория — это хорошо, но часто неправильно — результаты тестирования реализаций намного точнее. Пробуйте реализовать свои идеи. Проверяйте, какой будет реальная производительность. СУБД предлагают тысячи средств реализации. Нет единственного "лучшего решения" (если бы оно было, то поставщик СУБД только бы его и предоставлял). Иногда фрагментация данных повышает производительность, иногда — нет. Иногда использование компонента **interMedia Text** может повысить скорость поиска, а иногда — не повысит. Иногда хеш-кластер — лучшее решение, иногда от него нет никакого проку. В СУБД нет "вредных" средств (которых надо избегать любой ценой). Аналогично, нет и "панацеи", средств, решающих все проблемы.

Прежде чем утвердить именно такой проект для описанного выше приложения, мы опробовали несколько альтернативных подходов: пытались использовать *быстрый полный просмотр* индекса по функции (тоже быстро, но не настолько), применяли компонент **interMedia Text** (не помог, поскольку требовался поиск по шаблону типа **%ABC%**), пробовали добавить еще одно поле в таблицу **EMPLOYEES** (но она не вмещалась в буферный кэш, т.е. оказалась слишком большой для эффективного полного просмотра). Может показаться странным, что столько времени было потрачено на одну эту деталь реализации. Однако представленный выше запрос выполнялся от 150000 до 250000 раз в день, т.е. два-три раза в секунду в течение целого дня (при равномерном поступлении запросов). Но это предположение неверно: как и в большинстве систем, мы наблюдали пики активности. Если бы один этот запрос выполнялся медленно, вся система развалилась бы — и это всего лишь один из тысяч поддерживаемых запросов. Определив заранее предполагаемые слабые места или наиболее очевидные цели и сконцентрировав на них усилия, мы смогли создать хорошо масштабируемое приложение. Если бы использовался подход "настройка постфактум" приложение пришлось бы переписывать.

Применяйте защитное программирование

Снабдите код средствами отладки и настройки и оставьте их в производственной версии. Речь идет о способах трассировки действий приложения "извне". Установка **SQL_TRACE** (более детально рассматриваемая далее в этой главе) — это средство отладки и настройки. Система фиксации событий (EVENT) в СУБД Oracle — это тоже средство отладки и настройки (пример ее использования в Oracle представлен ниже). СУБД Oracle включает многочисленные средства отладки и настройки, чтобы разработчики ядра СУБД могли определять причины проблем, связанных с производительностью, даже не выезжая к клиенту. В приложениях такие средства тоже необходимы. Единственный способ заставить что-то работать быстрее — понять, где происходит замедление. Если известно только, что процесс "работает медленно", настроить его производительность будет крайне сложно. Если же процесс обильно снабжен соответствующими средствами отладки и трассировки и может по требованию регистрировать происходящие в нем события, вы сможете легко понять, что именно выполняется медленно.

Проверяйте производительность

Периодическая проверка производительности по ходу реализации принципиально важна. То, что нормально работает для 10 пользователей, не проходит для 100 или 1000. Проверка в реальных условиях — единственный способ убедиться, что можно будет достичь поставленных целей.

Главное — с первого же дня определить критерии (метрики) производительности. Это мой ночной кошмар: необходимо проверить и настроить производительность, а цель проста: "чтобы работало как можно быстрее". Такой настройкой производительности можно заниматься весь остаток жизни. Все может работать немного быстрее. Если цель ставится "как можно быстрее", она недостижима. Необходимо установить ограничения и вести разработку в рамках этих ограничений. Кроме того, если единственное ограничение — "как можно быстрее", можно делать и "как можно медленнее". Сравнивать не с чем, поэтому любая реализация получается достаточно быстродействующей. Последнее утверждение многих удивит: "как можно быстрее", значит, "так медленно, как получится"? Чтобы этого не случилось, надо установить четкие критерии.

Сначала надо проверить производительность изолированно. Если требуемая скорость работы не достигается в однопользовательском режиме, в реальной эксплуатации все будет еще медленнее. Запишите полученные результаты и сравнивайте их с предыдущими и последующими проверками. Так намного проще будет найти модуль, который раньше работал одну секунду, а теперь выполняется минуту.

Следующий шаг — проверить масштабируемость и протестировать приложение (или заложенные в него идеи) под предполагаемой нагрузкой, чтобы убедиться, что создаваемое решение будет масштабироваться. При тестировании многие проверяют функциональность. Я проверяю масштабируемость. Теперь, когда все модули приложения впервые собраны вместе, необходимо потратить время и усилия на обеспечение его масштабируемости и проверить, будет оно "летать" или нет. Именно в ходе этой проверки будут выявлены запросы, не использующие связываемые переменные, обнаружены блокирования и конфликты, создаваемые приложением, а также недостатки архитектуры. В ходе проверки масштабируемости эти проблемы становятся до боли очевидными. Если вы хотите успешно внедрить приложение, проверяйте его масштабируемость перед внедрением.

Я все время слышу обращения типа: "При разработке мы использовали подмножество реальных данных, и все было отлично. При внедрении системы в производственной среде все стало работать слишком медленно. Помогите нам, пожалуйста!". Единственное, что в этой ситуации можно сделать быстро, — убрать систему с производственного сервера и вернуться к расчетам. Необходимо разрабатывать систему и проверять ее на реальных объемах данных, с которыми она будет использоваться. Полный просмотр 100 строк на машине разработчика проходит успешно. Проблема производительности появляется, когда просматривается 100000 строк, причем одновременно 100 пользователями. Необходимо при разработке использовать реальные данные, реальные таблицы, реальную защиту — все, как будет в действительности. Это единственный способ сразу избавиться от "неэффективных" запросов. Не забудьте записывать результаты проверок производительности — сравнивая их, вы быстро найдете, что именно отрицательно повлияло на производительность. Вы также сможете доказать, что какое-то программное

или аппаратное решение повысило пропускную способность на столько-то — гадать не придется.

Например, следующая последовательность операторов сама по себе работает отлично:

```
declare
   l_rec  t%rowtype;
begin
   select * from T into l_rec from T where rowmuro = 1 FOR UPDATE;
   process(l_rec);
   delete from t where t.pk = l_rec.pk;
   commit;
end;
```

Она очень быстро обрабатывает первую запись. Измерив время работы в изолированной среде, вы можете сказать: "Я могу выполнять 5 транзакций в секунду (TPS)". Затем вы экстраполируете это значение и продолжаете: "При запуске 10 таких процессов, можно будет достичь 50 TPS". Проблема в том, что при запуске 10 процессов вы будете обрабатывать те же 5 транзакций в секунду (хорошо, если — 5), поскольку все они будут выполняться последовательно, один за другим, так как блокируют первую запись, а в каждый момент времени это может сделать лишь один процесс. Хорошо, если удается выполнять что-то быстро в изолированной среде, но добиться быстрого выполнения того же самого в многопользовательской среде — принципиально другая задача.

Даже если система используется только для чтения, другими словами, блокирование, вроде представленного выше, невозможно, масштабируемость надо проверять. Запросы требуют ресурсов — буферов, ввода/вывода с диска, процессорного времени для сортировки данных и т.д.; все, что требует ресурсов, необходимо проверять. Блокировка строки в рассмотренном выше примере — лишь один из типов ресурсов, а соревноваться приходится за сотни ресурсов, независимо от типа создаваемой системы.

Наконец, **последним шагом в процессе настройки** должна быть настройка СУБД. Большинство разработчиков ищут то, что я называю параметром инициализации **FAST=TRUE,** — какую-то простую установку, включив которую в файл инициализации, можно заставить все работать быстрее. **Такой установки нет.** Настройка СУБД, по моему опыту, дает наименьший рост производительности, если **уже используются разумные** установки. Крайне редко встречаются случаи, когда экземпляр настроен настолько плохо, что его настройка дает заметный рост производительности, но так бывает действительно редко (например, в буферном кэше хранилища данных сконфигурировано 300 блоков, а надо — 30000 или даже 300000). Настройка приложения, в частности изменение структур базы данных и реализация более производительных алгоритмов, — наиболее плодотворный путь, дающий максимально ощутимые результаты. Часто для решения проблемы почти ничего или вообще ничего нельзя сделать на уровне настройки экземпляра.

Теперь вернемся в реальный мир, где приложения просто создаются и не проверяются на масштабируемость, не имеют критериев производительности при разработке и вообще не снабжаются средствами отладки и тестирования. Что с этим делать (кроме как сбежать подальше, чтобы ничего не слышать об этом)? В некоторых случаях для диагностики и решения проблем можно будет использовать ряд существующих средств.

Во многих других случаях необходимо снабдить средствами трассировки сам код, особенно в больших приложениях, состоящих из нескольких взаимодействующих модулей. При поиске проблем в больших приложениях сложно определить даже, с чего начать. Если имеется клиент на Java, взаимодействующий с сервером приложений, который вызывает объект CORBA, изменяющий базу данных, то найти, что именно срабатывает медленно, непросто (если только приложение не предлагает соответствующих средств). Даже после выяснения причины, устранить ее очень сложно. Зачастую самое простое решение — правильное. Чем меньше составных частей необходимо учитывать, тем проще настраивать.

Связываемые переменные и разбор (еще раз)

Мы уже несколько раз с разных точек зрения касались темы связываемых переменных. Мы видели, например, что если не использовать связываемые переменные, до 90 процентов общего времени работы может уходить на разбор запросов, а не на их выполнение. Мы видели, как это может повлиять на разделяемый пул — ценнейший ресурс в Oracle. К этому моменту вы уже понимаете, что связываемые переменные определяют производительность системы. Не используйте их, и система будет работать во много раз медленнее, чем могла бы; кроме того, сократится количество одновременно поддерживаемых пользователей. Используйте их, и жизнь станет намного проще. По крайней мере, не придется возвращаться и исправлять программы, чтобы их можно было использовать.

Связываемые переменные важны, поскольку одной из особенностей конструкции СУБД Oracle является максимально возможное повторное использование планов оптимизатора. При получении SQL-оператора или блока PL/SQL, СУБД Oracle сначала просматривает разделяемый пул в поисках точно такого же. Например, в случае SQL-запроса сервер Oracle будет искать, не был ли уже такой запрос разобран и оптимизирован. Если запрос найден и план его выполнения можно использовать повторно, все готово для выполнения. Если же запрос не найден, сервер Oracle должен пройти через трудный процесс полного разбора запроса, оптимизации плана выполнения, проверки защиты и т.д. Это не только требует существенных вычислительных ресурсов процессора (обычно во много раз больше, чем собственно выполнение запроса), но и приводит к блокированию частей библиотечного кэша на сравнительно продолжительные периоды времени. Чем больше сеансов разбирают запросы, тем дольше приходится ждать освобождения защелок в библиотечном кэше, и работа системы постепенно останавливается.

Связываемые переменные и их использование — хороший пример того, почему необходимо проверять масштабируемость. В однопользовательской системе постоянный разбор запросов, не использующих связываемые переменные, можно и не заметить. Единственный сеанс будет, конечно, работать медленнее, чем мог бы, но все же "достаточно быстро". А вот при одновременном запуске 10 или 100 таких сеансов, система неизбежно остановится. Избыточно тратятся ценные ресурсы (время процессора, библиотечный кэш, защелки библиотечного кэша). Применяя же связываемые переменные, можно уменьшить использование этих ресурсов во много раз.

Я собираюсь еще раз продемонстрировать огромное влияние связываемых переменных на производительность. В главе 1 я показывал это на примере одного сеанса: если

не используются связываемые переменные, работа приложения замедляется. Там мы видели, что блок кода без связываемых переменных выполняется 15 секунд. Тот же код, но написанный с использованием связываемой переменной выполняется за 1,5 секунды. Здесь я продемонстрирую последствия неиспользования связываемых переменных в многопользовательской среде. Уже понятно, что без связываемых переменных код работает медленнее; теперь давайте оценим, как их отсутствие повлияет на масштабируемость.

Для этого теста я буду использовать следующие таблицы. Обратите внимание: для выполнения этого примера необходим доступ к представлению **V$SESSION_EVENT,** т.е. наличие привилегии SELECT для представления **V$SESSION_EVENT.** Кроме того, необходимо установить параметр системы **(SYSTEM)** или сеанса **(SESSION) TIMED_STATISTICS,** чтобы получать осмысленные результаты (иначе время выполнения каждого оператора будет равно нулю). Это можно сделать с помощью оператора **ALTER SESSION SET TIMED_STATISTICS=TRUE.** Начнем с создания глобальной временной таблицы **SESS_EVENT,** которая будет использоваться сеансом для хранения "предыдущих значений" событий, наступления которых ожидал сеанс. Эта таблица SESS_EVENT будет использоваться для определения ожидаемых сеансами событий, количества ожиданий и времени ожидания в сотых долях секунды.

```
tkyte@TKYTE816> create global temporary table sess_event
  2  on commit preserve rows
  3  as
  4  select * from v$session_event where 1=0;

Table created.
```

Теперь создадим "прикладную" таблицу для тестирования:

```
tkyte@TKYTE816> create table t
  2 (c1  int, c2 int, c3 int, c4 int)
  3 storage (freelists 10);

Table created
```

Я хочу проверить, что будет происходить при одновременной вставке строк в эту таблицу несколькими пользователями. В главе 6 было показано, как может повлиять наличие нескольких списков свободных мест (freelists) на одновременную вставку, поэтому соответствующая установка уже включена в оператор создания таблицы. Теперь определим, наступления каких событий будет ожидать наше "приложение". Для этого сделаем копию набора текущих ожидаемых событий сеанса, выполним блок кода, который необходимо проанализировать, а затем вычислим продолжительность ожиданий, имевших место при выполнении этого блока кода:

```
tkyte@TKYTE816> truncate table sess_event;

Table truncated.

tkyte@TKYTE816> insert into sess_event
  2  select * from v$session_event
  4  where sid = (select sid from v$mystat where rownum = 1) ;

3 rows created.
```

```
tkyte@TKYTE816> declare
  2        l_number number;
  3   begin
  4        for i in 1 .. 10000
  5        loop
  6             l_number := dbms_random.random;
  7
  8             execute immediate
  9             'insert into t values (' If l_number || ',' ||
 10                                        l_number || ',' ||
 11                                        l_number || ',' ||
 12                                       l_number || ')' ;
 13        end loop;
 14        commit;
 15   end;
 16   /

PL/SQL procedure successfully completed

tkyte@TKYTE816> select a.event,
  2         (a.total_waits-nvl(b.total_waits,0)) total_waits,
  3      (a.time_waited-nvl(b.time_waited,0))     time_waited
  4      from (select *
  5              from v$session_event
  6             where sid = (select sid from v$mystat where rownum = 1)) a,
  7           sess_event b
  8    where a.event = b.event(+)
  9       and (a.total waits-nvl(b.total_waits,0)) > 0
 10   /

EVENT                          TOTAL_WAITS     TIME_WAITED
------------------------------ ------------ ---------------
SQL*Net message from client              4              14
SQL*Net message to client                5               0
log file sync                            5               2
```

В этом небольшом тесте мы создаем **уникальный** оператор INSERT, который будет выглядеть примерно так:

```
insert into t values (12323, 12323, 12323, 12323);
insert into t values (632425, 632425, 632425, 632425);
```

Представленные выше результаты получены в однопользовательском режиме. Если выполнить это одновременно в двух сеансах, увидим примерно такой отчет о времени ожидания:

```
EVENT                          TOTAL_WAITS     TIME_WAITED

SQL*Net message from client              4              18
SQL*Net message to client                5               0
enqueue                                  2               0
latch free                             142             235
log file sync                            2               2
```

Как видите, сеанс много раз ждал освобождения защелки (причем суммарное время ожидания — достаточно большое). Кроме того, наблюдалось ожидание следующих событий:

- **SQL*Net message from client.** Сервер ждал, пока клиент пошлет ему сообщение. В данном случае клиент — SQL*Plus. В большинстве случаев ожидание этого события можно игнорировать, но если при выполнении приложения предполагаются длительные раздумья пользователя, это число неизбежно будет большим. В нашем случае сеанс SQL*Plus постоянно выдавал операторы на сервер, поэтому время ожидания должно быть небольшим. Если бы оно было большим, проблема была бы связана с клиентом (узким местом был бы клиент, неспособный достаточно часто обращаться к базе данных).
- **SQL*Net message to client.** Сколько времени потребовалось на передачу сообщений с сервера клиенту (SQL*Plus).
- **Enqueue.** Ожидание той или иной блокировки.
- **Log file** sync. Время ожидания сброса буфера журнала повторного выполнения на диск процессом **LGWR** при фиксации.

Все события, которых может ожидать сервер, описаны в руководстве "Oracle8i Reference Manual" в приложении "Oracle Wait Events". В этом руководстве содержатся подробные описания всех событий, которые можно увидеть в представлении V$SESSION_EVENT.

На ожидание события освобождения защелки в данном случае надо обратить внимание. Эта защелка предотвращала одновременный доступ к разделяемой области SQL. Я предполагал это ожидание с учетом характеристик выполнявшейся транзакции. Другие запросы к представлениям V$ могут подтвердить это (далее мы рассмотрим представления V$ более детально). Поскольку теперь два сеанса выполняют "жесткий разбор" (анализируют запрос, никогда ранее не обрабатывавшийся), появляются конфликты доступа к разделяемой области SQL. Двум сеансам надо изменить общую структуру данных, и делать это они могут только поочередно. В следующей таблице показано количество ожиданий освобождения защелки для 1, 2, 3, 4 и 5 сеансов, одновременно выполняющих представленную выше транзакцию:

	1 пользователь	***2 пользователя***	***3 пользователя***	***4 пользователя***	***5 пользователей***
Количество ожиданий	0	102	267	385	542
Время (секунд)	0	1.56	5.92	10.72	16.67

Учтите, что в таблице представлена информация для одного сеанса: каждому сеансу пришлось ждать столько раз и такое суммарное количество времени. В случае двух пользователей ждать пришлось три секунды, трех — около 18, четырех — около 40 и т.д. В какой-то момент при добавлении дополнительных пользователей **ожидать** мы будем

дольше, чем работать. Чем больше добавляется пользователей, тем дольше приходится ждать, и эти ожидания могут оказаться настолько продолжительными, что добавление пользователей не только замедлит ответы на запросы, но и снизит общую пропускную способность сервера в целом. Как это исправить? Просто переписать блок кода с использованием связываемых переменных, например, так:

```
tkyte@TKYTE816> declare
  2        l_number number;
  3  begin
  4       for i in 1 .. 10000
  5        loop
  6             l_number := dbms_random.random;
  7
  8             execute immediate
  9            'insert into t values (:x1, :x2, :x3, :x4)'
 10                      using l_number, l_number, l_number, l_number;
 11       end loop;
 12       commit;
 13 end;
 14 /

PL/SQL procedure successfully completed.
```

Если собираетесь выполнять этот пример в Oracle 8.1.5, см. информацию об ошибках на Web-сайте издательства Wrox, ***http://www.wrox.com.*** *Там можно узнать, что нужно сначала вызвать процедуру* ***dbms_random.initialize.*** *Кроме того, для установки пакета* ***dbms_random*** *в Oracle 8i вплоть до версии 8.1.6, необходимо от имени пользователя SYS выполнить сценарий* ***catoctk.sql,*** *который находится в каталоге* ***$ORACLE_HOME/rdbms/admin.***

В этом случае мы видим заметное улучшение:

	1 пользователь	*2 пользователя*	*3 пользователя*	*4 пользователя*	*5 пользователей*
Количество ожиданий	0	47 (46%)	65 (25%)	89 (23%)	113 (20%)
Время (секунд)	0	0.74 (47%)	1.29 (21%)	2.12 (19%)	3.0 (17%)

Это существенное улучшение, но можно сделать еще лучше. При двух пользователях общее количество ожиданий освобождения защелок в обшей области SQL сократилось до 46 процентов исходного значения, как и время ожидания. При добавлении пользователей ситуация еще улучшается. В случае пяти пользователей количество ожиданий сократилось на 80 процентов, и суммарное время ожиданий составило лишь 17 процентов исходного времени.

Однако я утверждаю, что можно пойти еще дальше. В представленном выше примере мы избегали жесткого разбора, но постоянно выполняли "мягкий разбор". На каждой итерации цикла необходимо искать оператор INSERT в разделяемом пуле и прове-

рять, можно ли его использовать. Представленный выше подход на базе оператора **EXECUTE IMMEDIATE** можно описать так:

```
loop
   разобрать
   связать
   выполнить
   закрыть
end;
```

А лучше было бы так:

```
разобрать
loop
   связать
   выполнить
end;
закрыть;
```

В нашем случае для этого можно использовать статический SQL в PL/SQL-блоке либо пакет **DBMS_SQL,** позволяющий процедурно формировать и выполнять SQL-операторы (примеры и особенности использования пакета **DBMS_SQL** представлены в главе 16). Я буду использовать статический SQL со связываемыми переменными следующим образом:

```
tkyte@TKYTE816> declare
  2       l_number number;
  3  begin
  4      for i in 1 . . 10000
  5       loop
  6           l_number := dbms_random.random;
  7
  8           insert into t
  9                   values(l_number, l_number, l_number, l_number);
 10       end loop;
 11       commit;
 12  end;
 13  /

PL/SQL procedure successfully completed.
```

PL/SQL будет автоматически кэшировать курсор — это одно из основных преимуществ PL/SQL. Оператор вставки обычно будет разобран только один раз на сеанс, если он помещен в хранимую процедуру. Он будет разобран один раз при выполнении блока, если помещен в анонимный блок. Теперь можно снова выполнить тестирование:

	1 пользователь	***2 пользователя***	***3 пользователя***	***4 пользователя***	***5 пользователей***
Количество ожиданий	0	1	1	*1*	*7*
Время (секунд)	0	0	0.01	0.10	0.08

Время ожидания освобождения защелок крайне мало — им просто можно пренебречь. Рассмотренный пример показывает, почему:

- использование связываемых переменных определяет производительность;
- важно избегать лишних мягких разборов запроса.

Сокращение количества мягких разборов оператора SQL в некоторых случаях, как показано выше, дает даже больше, чем использование связываемых переменных. Не торопитесь закрывать курсор: если его можно будет использовать повторно, расходы ресурсов на поддержку курсора в открытом состоянии в ходе выполнения программы с лихвой покрываются повышением производительности.

В Oracle 8.1.6 добавлена новая возможность: совместное использование курсора — **CURSOR_SHARING.** Совместное использование курсора — это своего рода автоматическая подстановка связываемых переменных. Сервер вынужден переформулировать поступающий запрос так, чтобы в нем использовались связываемые переменные, прежде чем разбирать его. В результате, запрос вида:

```
scott@TKYTE816> select * from emp where ename = 'KING';
```

будет автоматически преобразован в вид:

```
select * from emp where ename =:SYS_B_0
```

или, в версиях 8.1.6 и 8.1.7

```
select * from emp where ename =:"SYS_B_0"
```

Это шаг в правильном направлении, но нельзя использовать этот параметр как окончательное и долговременное решение проблемы. Использование **CURSOR_SHARING** имеет побочные эффекты, о которых надо помнить. Производительность плохо написанной программы может существенно увеличиться при установке **CURSOR_SHARING= FORCE,** но работать она все равно будет медленнее, чем могла бы, да и масштабируемость будет ограничена. Как было показано выше, количество и время ожиданий можно значительно сократить за счет использования связываемых переменных. Того же результата можно добиться с помощью установки **CURSOR_SHARING=FORCE.** Однако пока не предотвращены избыточные мягкие разборы операторов, от всех ожиданий избавиться нельзя. Совместное использование курсоров не избавляет от лишних мягких разборов. Если установка параметра **CURSOR_SHARING** дает существенный результат, значит, вы требуете от сервера Oracle слишком часто разбирать большое количество запросов. Если требуется частый разбор большого количества запросов, то установка **CURSOR_SHARING** решит проблему связываемых переменных, но не устранит дополнительный расход ресурсов на повторный мягкий разбор. Хотя и не так очевидно, как отсутствие связываемых переменных, большое количество мягких разборов тоже ограничивает производительность и масштабируемость. Единственное решение — использовать связываемые переменные и по возможности использовать курсоры повторно. Например, если бы я создавал программу на Java, я никогда не написал такой фрагмент:

```
String getWordb(int ID, int IDLang, Connection conn) throws SQLException {
   CallableStatement stmt = null;
   stmt = conn.prepareCall("{ call get.wordb (?,?,?)}");
   stmt.setInt(1,ID);
   stmt.setlnt(2,IDLang);
   stmt.registerOutParameter (3, Java.sql.Types.VARCHAR);
   stmt.execute();
   String word = stmt.getstring (3) ;
   stmt.close();
   return word;
}
```

Я бы написал так:

```
CallableStatement stmt = null;
String getWordbfint ID, int IDLang, Connection conn)
throws SQLException {
   if (stmt — null) {
      stmt = conn.prepareCall("{call get.wordb (?,?,?))");
      stmt.registerOutParameter (3, Java.sql.Types.VARCHAR);
   }
   stmt.setlnt(1,ID);
   stmt.setlnt(2,IDLang);
   stmt.execute();
   return stmt.getstring (3);
}
```

Здесь я гарантирую использование связываемых переменных, подставляя символы-заместители в оператор. Кроме того, я разбираю оператор не более одного раза при каждом выполнении программы. Это дает огромный выигрыш в производительности. В ходе одного тестирования я вызвал "плохую" и "хорошую" реализацию по 1000 раз. Плохая реализация, в которой мягкий разбор выполнялся при каждом обращении, работала две с половиной секунды. Хорошая реализация работала одну секунду. Это в однопользовательском режиме. Как вы уже знаете, при добавлении дополнительных пользователей, каждый из которых выполняет тысячи мягких разборов, работа существенно замедлится за счет ожидания снятия защелок в ходе мягкого разбора. Этих дополнительных расходов можно и нужно избегать в создаваемых приложениях.

Теперь ненадолго вернемся к параметру **CURSOR_SHARING.** Я уже говорил, что имеются побочные эффекты использования параметра **CURSOR_SHARING,** о которых надо знать. Их можно разбить на следующие категории:

- **Проблемы оптимизатора.** При установке **CURSOR_SHARING** из запроса удаляются **все** символьные строки и числовые константы; у оптимизатора остается меньше информации для работы. Это может привести к совершенно другим планам выполнения запросов.

- **Проблемы с результатами выполнения запросов.** Длина извлекаемых столбцов может неожиданно измениться. Запросы, обычно возвращавшие данные типа **VARCHAR2(5)** и **NUMBER(2),** могут теперь возвращать **VARCHAR2(30)** и **NUMBER(5).** Фактический размер возвращаемых данных не изменится, но приложение будет получать информацию о том, что столбец может содержать строки длиной до 30 байт, а это может повлиять на отчеты и создающие их приложения.
- **Сложности в оценке планов выполнения запросов.** Они возникнут из-за того, что **EXPLAIN PLAN** будет "видеть" не такой запрос, как поступает в базу данных. Это усложняет настройку производительности запросов. Средства типа **AUTOTRACE** в **SQL*Plus** становятся ненадежными источниками информации при установке параметра **CURSOR_SHARING.**

Мы подробно рассмотрим каждую из этих проблем, но сначала давайте поговорим о проблемах оптимизатора. Они непосредственно влияют на производительность приложения. Рассмотрим следующий простой пример, в котором выполняется запрос, содержащий как связываемые переменные, так и строковые константы. Производительность этого запроса до установки параметра **CURSOR_SHARING** — отличная. Производительность после включения параметра **CURSOR_SHARING** — ужасная. Причина падения производительности в том, что оптимизатор теперь имеет намного меньше информации, необходимой ему для работы. По строковым константам оптимизатор мог определить, что один индекс будет более избирателен по сравнению с другим. При удалении всех констант оптимизатор не может прийти к такому выводу. Подобная проблема — не редкость для многих приложений. В большинстве приложений в операторах SQL используются и связываемые переменные, и константы. Не использовать связываемые переменные — это ужасно, но использовать связываемые переменные **всегда** — немногим лучше. **Вот** какую таблицу надо создать для данного примера:

```
tkyte@TKYTE816> create table t as
  2  select * from all_objects;

Table created.

tkyte@TKYTE816> create index t_idx1 on t(OBJECT_NAME);

Index created.

tkyte@TKYTE816> create index t_idx2 on t(OBJECT_TYPE);

Index created.

tkyte@TKYTE816> analyze table t compute statistics
  2  for all indexed columns
  3  for table;

Table analyzed.
```

По таблице имеется два индекса. Индекс по столбцу **OBJECT_TYPE** используется для выполнения представленного ниже запроса. Индекс по столбцу **OBJECT_NAME** интенсивно используется другим приложением. Оба эти индекса необходимы. Мы будем выполнять к таблице следующий запрос:

```
select *
  from t
 where object_name like :search_str
   and object_type in ('FUNCTION', 'PROCEDURE', 'TRIGGER');
```

Этот запрос создавался для заполнения выпадающего списка в приложении. Пользователь выбирает из списка процедуру, функцию или триггер для редактирования. Запрос выполняется тысячи раз в день.

Если посмотреть на реальные данные в таблице:

```
tkyte@TKYTE816> compute sum of cnt on report
tkyte@TKYTE816> break on report
tkyte@TKYTE816> select object_type, count(*) cnt from t group by
  2  object_type;

OBJECT_TYPE             CNT
----------------------------
CONSUMER GROUP            2
CONTEXT                   4
DIRECTORY                 1
FUNCTION                 27
INDEX                   301
INDEXTYPE                 3
JAVA CLASS             8926
JAVA DATA               288
JAVA RESOURCE            69
JAVA SOURCE               4
LIBRARY                  33
LOB                       3
OPERATOR                 15
PACKAGE                 229
PACKAGE BODY            212
PROCEDURE                24
SEQUENCE                 35
SYNONYM                9817
TABLE                   292
TRIGGER                   7
TYPE                    137
TYPE BODY                12
UNDEFINED                 1
VIEW                   1340
----------------------------
                      21782

24 rows selected.
```

то окажется, что конструкция IN вернет 58 строк, а количество строк, возвращаемых конструкцией LIKE, определить заранее нельзя (их может быть сколько угодно — от 0 до 21782). Если выполнить в SQL*Plus запрос следующего вида:

```
tkyte@TKYTE816> variable search_str varchar2(25)
tkyte@TKYTE816> exec :search_str := '%';

PL/SQL procedure successfully completed.
```

```
tkyte@TKYTE816> set autotrace traceonly
tkyte@TKYTE816> select * from t t1 where object_name like :search_str
  2  and object_type in('FUNCTION', 'PROCEDURE', 'TRIGGER');

58 rows selected.


Execution Plan

   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=3 Bytes=291)
   1    0   INLIST ITERATOR
   2    1    TABLE ACCESS (BY INDEX ROWID) OF 'T' (Cost=5 Card=3 Bytes=291)
   3    2   INDEX (RANGE SCAN) OF 'T_IDX2' (NON-UNIQUE) (Cost=2 Card=3)

Statistics

       222  recursive calls
         0  db block gets
        45  consistent gets
         3  physical reads
         0  redo size
      6930  bytes sent via SQL*Net to client
       762  bytes received via SQL*Net from client
         5  SQL*Net roundtrips to/from client
         1  sorts (memory)
         0  sorts (disk)
        58  rows processed
```

то с учетом того, что мы знаем о распределении данных, кажется очевидным, что оптимизатор должен использовать индекс по столбцу **OBJECT_TYPE** для выборки 58 строк из 21000, а затем применять к этим строкам конструкцию **LIKE**. Это показывает, что иногда использование констант оправдано. В данном случае использовать связываемую переменную не стоит — лучше указать константу. При этом оптимизатор сможет определить, что запрос будет выполняться быстрее при использовании конкретного индекса. Если же сделать так:

```
tkyte@TKYTE816> alter session set cursor_sharing = force;

Session altered.

tkyte@TKYTE816> select * from t t2 where object_name like :search_str
  2  and object_type in('FUNCTION', 'PROCEDURE', 'TRIGGER');

58 rows selected.


Execution Plan

   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=3 Bytes=291)
   1    0   INLIST ITERATOR
   2    1    TABLE ACCESS (BY INDEX ROWID) OF 'T' (Cost=5 Card=3 Bytes=291)
   3    2      INDEX (RANGE SCAN) OF 'T_IDX2' (NON-UNIQUE) (Cost=2 Card=3)
```

```
statistics

          0  recursive calls
          0  db block gets
      19256  consistent gets
        169  physical reads
          0  redo size
       7480  bytes sent via SQL*Net to client
        762  bytes received via SQL*Net from client
          5  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
         58  rows processed
```

то, хотя выдаваемый план выполнения запроса якобы не изменился **(AUTOTRACE** показывает точно такой же план), различие в количестве **consistent gets** (логических чтений) получается существенное, так что явно что-то изменилось. Сервер фактически выполняет запрос:

```
select *
  from t t2
 where object_name like :search_str
  and object_type in( :SYS_B_0,:SYS_B_1, :SYS_B_2 )
```

и уже не может определить, сколько строк он будет выбирать через индекс по столбцу **OBJECT_TYPE**. Этот пример также показывает, как установка параметра **CURSOR_SHARING** затрудняет настройку. Выдаваемый утилитой SQL*Plus план запроса заставляет думать, что считывается индекс **T_IDX2,** но если посмотреть на количество **consistent gets** (логических чтений), окажется, что их выполнено 19256. Первый запрос, действительно использующий индекс **T_IDX2,** обработал 45 блоков. В данном случае установка **autotrace** выдает нам некорректный план. Утилита SQL*Plus не знает, какой запрос в действительности выполняется. Я включил трассировку **SQL_TRACE** (подробнее об этом — в следующем разделе), и теперь для двух запросов можно четко увидеть различие:

```
select * from t t1 where object_name like :search_str
and object_type in('FUNCTION', 'PROCEDURE', 'TRIGGER')
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	5	0.01	0.09	14	34	0	58
total	7	0.01	0.09	14	34	0	58

```
Rows        Row  Source Operation
-------------------------------------------------------
        58   INLIST ITERATOR
        58    TABLE ACCESS BY INDEX ROWID T
        61     INDEX RANGE SCAN (object id 25244)
```

```
select * from t t2 where object_name like :search_str
and object_type in(:SYS_B_O,:SYS_B_1, :SYS_B_2)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        5      0.15       1.77        255      19256          0          58
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        7      0.15       1.77        255      19256          0          58

Rows     Row Source Operation
-------  ---------------------------------------------------
     58  TABLE ACCESS BY INDEX ROWID T
  21783   INDEX RANGE SCAN (object id 25243)
```

Средства SQL_TRACE и TKPROF могут показать, что происходит на самом деле. Второй запрос был выполнен с использованием другого индекса (идентификатор объекта — другой), что в данном случае не оптимально. Наш запрос выполняется в 15-20 раз дольше и обрабатывает огромный объем данных. Эта проблема ненужных связываемых переменных будет проявляться во многих приложениях, намеренно использующих как связываемые переменные, так и константы. Без дополнительной информации, обеспечиваемой константами в запросе, оптимизатор может принять неверное решение. Только при корректном использовании связываемых переменных, где только возможно, и констант — там, где это обязательно, можно достичь сбалансированности. Установка **CURSOR_SHARING** — временное решение, с которого можно начать, но по изложенным выше причинам оно не может быть долговременным.

Параметр **CURSOR_SHARING** на только сбивает с толку оптимизатор, как было показано выше, он может повлиять и на другие средства Oracle. Например, в главе 7 описывалось такое средство как *индексы по функции*. По сути, в Oracle можно создать индекс по функции. Используя пример из той главы, где индекс создавался следующим образом:

```
tkyte@TKYTE816> create index test_soundex_idx on
  2                test_soundex(substr(my_soundex(name),1,6))
  3  /

Index created.
```

можно выяснить, что установка параметра CURSOR_SHARING не позволит для выполнения запроса

```
tkyte@TKYTE816> select name
  2      from test_soundex C
  3     where substr(my_soundex(name),1,6)  = ray_soundex('FILE$')
  4  /
```

использовать этот индекс, поскольку литералы 1 и 6 будут заменены связываемыми переменными. В данном случае можно решить эту проблему, "спрятав" константы в представлении, но ее все равно надо учитывать. Не будет ли других "неожиданных" отличий?

Еще один побочный эффект установки параметра **CURSOR_SHARING** —возможные неожиданные изменения размера столбцов, возвращаемых запросом. Рассмотрим следующий пример, в котором выдается размер возвращаемых столбцов до и после установки **CURSOR_SHARING**. В этом примере для динамического разбора и описания запроса используется пакет **DBMS_SQL**. Он выдает размеры столбцов, которые сообщает приложению сервер Oracle:

```
tkyte@TKYTE816> declare
  2        l_theCursor      integer default dbms_sql.open_cursor;
  3        l_descTbl        dbms_sql.desc_tab;
  4        l_colCnt         number;
  5  begin
  6       execute immediate 'alter session set cursor_sharing=exact';
  7       dbms_output.put_line('Without Cursor Sharing:');
  8        for i in 1..2
  9       loop
 10            dbms_sql.parse(   l_theCursor,
 11                              'select substr(object_name, 1, 5) c1,
 12                                      55 c2,
 13                                     ''Hello" c3
 14                                from all_objects t'||i,
 15                              dbms_sql.native);
 16
 17            dbms_sql.describe_columns(l_theCursor,
 18                                      l_colCnt, l_descTbl);
 19
 20           for i in 1 .. l_colCnt loop
 21                dbms_output.put_line('Column ' ||
 22                                      l_descTbl(i).col_name ||
 23                                     ' has a length of ' ||
 24                                      l_descTbl(i).col_max_len);
 25            end loop;
 26            execute immediate 'alter session set cursor_sharing=force';
 27           dbms_output.put_line('With Cursor Sharing:');
 28       end loop;
 29
 30      dbms_sql.close_cursor(l_theCursor);
 31       execute immediate 'alter session set cursor_sharing=exact';
 32  end;
33  /
Without Cursor Sharing:
Column C1 has a length of 5
Column C2 has a length of 2
Column C3 has a length of 5
With Cursor Sharing:
Column C1 has a length of 30
Column C2 has a length of 22
Column C3 has a length of 32

PL/SQL procedure successfully completed.
```

Размер первого столбца с 5 байт вырос до 30, потому что функция SUBSTR(OBJECT_NAME, 1, 5) была переписана как SUBSTR(OBJECT_NAME, :SYS_B_0, :SYS_B_l) Сервер "не знает", что функция может вернуть максимум 5 байт, поэтому теперь возвращается значение 30 (длина поля OBJECT_NAME). Длина второго столбца выросла с 2 до 22 байт, потому что сервер больше "не знает", что будет возвращено значение 55 — известно только, что будет возвращаться число, а длина числа может составлять до 22 байт. Для последнего столбца выдано стандартное значение — если бы строка HELLO была больше, то и стандартное значение оказалось бы больше (например, если бы использовалась 35-байтовая строка, стандартное значение составляло бы 128 байт).

Можно возразить: "Ну, длина возвращаемых данных не изменится, изменится только значение длины, выдаваемое сервером по запросу описания результирующего множества...". Проблемы возникнут во всех сценариях SQL*Plus, во всех отчетах, создаваемых с помощью различных инструментальных средств, и вообще во всех приложениях, запрашивающих у сервера характеристики результирующего множества для соответствующего форматирования столбцов. Результаты, выдаваемые этими приложениями, при установке параметра **CURSOR_SHARING** изменятся; форматирование тщательно подготовленных отчетов во многих случаях окажется неадекватным. Только подумайте, как это может повлиять на множество уже существующих приложений! Этот побочный эффект очень легко продемонстрировать:

```
tkyte@TKYTE816> select substr(object_name,1,2)
  2     from all_objects tl
  3    where rownum = 1
  4   /

SU

/1

tkyte@TKYTE816> alter session set cursor_sharing = force;

Session altered.

tkyte@TKYTE816> select substr(object_name,1,2)
  2     from all_objects t2
  3    where rownum = 1
  4   /

SUBSTR(OBJECT_NAME,1,2)

/1
```

Утилита SQL*Plus перешла от столбца из 2 символов к столбцу из 30. Это, несомненно, повлияет на ранее успешно работавшие отчеты.

Используются ли связываемые переменные?

Когда я спрашиваю: "Используете ли вы связываемые переменные?", в ответ получаю вопрос: "А как узнать, используются ли связываемые переменные?". К счастью, выяснить это весьма просто; вся необходимая информация находится в разделяемом пуле.

Я создал сценарий, который часто использую (и распространяю), для выявления операторов, которые совпадали бы при использовании связываемых переменных. Чтобы показать, как работает этот сценарий, я искусственно заполнил разделяемый пул операторами SQL, не использующими связываемых переменных:

```
tkyte@TKYTE816> create table t (x int) ;

Table created.

tkyte@TKYTE816> begin
  2       for i in 1 .. 100
  3       loop
  4           execute immediate 'insert into t values (' || i || ')' ;
  5       end loop;
  6  end;
  7  /

PL/SQL procedure successfully completed.
```

Теперь можно демонстрировать сценарий. Он начинается с создания функции, удаляющей константы из строк. Она будет преобразовывать SQL-операторы вида:

```
insert into t values ('hello', 55);
insert into t values ('world', 66);
```

в такой вид:

```
insert into t values ('#', @) ;
```

Все операторы, которые могли бы совпасть при использовании связываемых переменных, теперь легко выявить: оба представленных выше уникальных оператора INSERT после подстановки станут одинаковыми. Такое преобразование позволяет выполнить следующая функция:

```
tkyte@TKYTE816> create or replace
  2   function remove_constants(p_query in varchar2)
  3   return varchar2
  4   as
  5       l_query long;
  6       l_char  varchar2(1) ;
  7       l_in_quotes boolean default FALSE;
  8   begin
  9       for i in 1 .. length(p_query)
 10       loop
 11           l_char := substr(p_query,i,1);
 12           if (l_char = '''' and l_in_quotes)
 13           then
 14               l_in_quotes := FALSE;
 15           elsif (l_char = " " and NOT l_in_quotes)
 16           then
 17               l_in_quotes := TRUE;
 18               l_query := l_query || '''#';
 19           end if;
 20           if ( NOT l_in_quotes ) then
 21               l_query := l_query || l_char;
```

```
 22           end if;
 23       end loop;
 24       l_query := translate(l_query, '0123456789', '@@@@@@@@@@');
 25       for i in 0 .. 8 loop
 26           l_query := replace(l_query, lpad('@',10-i,'@'), '@');
 27         l_query := replace(l_query, lpad(' ',10-i,' ')/ ' ');
 28       end loop;
 29       return upper(l_query);
 30  end;
 31  /

Function created.
```

Для основного кода сценария мы сделаем копию представления **V$SQLAREA** — запросы к этому представлению выполняются долго, поэтому хотелось бы обращаться к нему только один раз. Мы копируем его во временную таблицу, чтобы можно было работать с соответствующими данными:

```
tkyte@TKYTE816> create global temporary table sql_area_tmp
  2  on commit preserve rows
  3  as
  4  select sql_text, sql_text sql_text_wo_constants
  5     from v$sqlarea
  6    where 1=0
  7  /

Table created.


tkyte@TKYTE816> insert into sql_area_tmp (sql_text)
  2  select sql_text from v$sqlarea
  3  /
```

436 rows created.

Пройдем по всем строкам этой таблицы и определим преобразованный **SQL_TEXT**, из которого удалены все константы:

```
tkyte@TKYTE816> update sql_area_tmp
  2      set sql_text_wo_constants = remove constants(sql_text);
  3  /

436 rows updated.
```

Теперь все готово для выявления "плохих" операторов SQL:

```
tkyte@TKYTE816> select sql_text_wo_constants, count(*)
  2     from sql_area_tmp
  3    group by sql_text_wo_constants
  4   having count(*) > 10
  5    order by 2
  6  /

SQL_TEXT_WO_CONSTANTS          COUNT(*)

INSERT INTO T VALUES (6)          100
```

Итак, в разделяемом пуле имеется 100 операторов **INSERT,** отличающихся только одним числовым полем в конструкции **VALUES.** Это **почти наверняка** означает, что кто-то забыл использовать связываемые переменные. Бывают вполне оправданные случаи наличия нескольких экземпляров SQL-оператора в разделяемом пуле (например, в базе данных может быть пять разных таблиц с именем Т). Выяснив, что разумной причины, объясняющей наличие нескольких экземпляров одного и того же оператора, нет, необходимо найти соответствующего разработчика, научить его, как делать правильно, и заставить исправить операторы. Я считаю это **ошибкой** в программе, а не вполне допустимым приемом, а ошибка должна быть исправлена.

Итак, в этом разделе мы обсуждали важность использования связываемых переменных в приложении, а также необходимость свести к минимуму количество разборов запросов. Было описано новое средство — параметр инициализации **CURSOR_SHARING,** который мог показаться панацеей при решении этих проблем, но оказалось, что все не так просто. Установка параметра **CURSOR_SHARING** может использоваться как временное, промежуточное решение определенных проблем производительности приложения, но достичь максимальной производительности и масштабируемости можно только за счет правильной реализации приложений.

Я не знаю, как еще подчеркнуть этот момент. Приходилось видеть множество систем, потерпевших неудачу из-за того, что их создатели не учитывали представленных выше фактов. Как я уже писал в самом начале книги, если бы мне пришлось писать книгу о том, как создавать немасштабируемые и медленно работающие приложения для СУБД Oracle, она содержала бы всего одну главу, где было бы сказано: "не используйте связываемые переменные". Использование связываемых переменных и правильных приемов разбора операторов еще не гарантирует масштабируемость, но если их не использовать, то ее точно не будет.

SQL_TRACE, TIMED_STATISTICS и TKPROF

SQL_TRACE, TIMED_STATISTICS и **TKPROF-** мои самые любимые инструментальные средства. Я использовал их множество раз для выявления причин неудовлетворительной производительности системы. Во всех этих случаях приходилось настраивать системы, которые я не создавал, поэтому и не знал, где искать. Эти установки и средства дают необходимую информацию для начала работы.

Если коротко, параметр **SQL_TRACE** включает регистрацию всех операторов SQL, выполняемых приложением, информации о производительности, полученной в ходе выполнения этих операторов SQL, и фактически использованных планов выполнения операторов. Как было показано в предыдущем разделе, посвященном параметру **CURSOR_SHARING, AUTOTRACE** показывает неверный план, а вот параметр **SQL_TRACE** и утилита **TKPROF** показывают именно тот план, который реально использован. **TIMED_STATISTICS** — это параметр, при установке которого сервер регистрирует продолжительность выполнения каждого шага. Наконец, **TKPROF** — это простая программа, используемая для преобразования файла трассировки в более удобочитаемый вид. В этом разделе я продемонстрирую, как использовать установку **SQL_TRACE** и утилиту **TKPROF,** и попытаюсь разъяснить значение содержимого ис-

пользуемых ими файлов. Я не столько буду описывать, как настроить конкретный запрос, сколько покажу, как с помощью этих средств найти запросы, требующие настройки. Более подробную информацию о настройке отдельных запросов можно найти в руководстве *Oracle8i Designing and Tuningfor Performance Manual.* Там описаны различные способы доступа к данным, использование подсказок оптимизатору для настройки запросов и т.д.

Параметр **TIMED_STATISTICS** управляет тем, будет ли сервер Oracle собирать информацию о времени выполнения различных действий в базе данных. Он может иметь одно из двух значений: **TRUE** или **FALSE.** Эта возможность настолько полезна, что я часто оставляю значение **TRUE,** даже когда не занимаюсь настройкой — влияние этого значения на производительность СУБД, как правило, незначительно (была проблема в Oracle 8.1.5, где совместное использование операторов SQL не всегда обеспечивалось, если параметр **TIMED_STATISTICS** имел значение **TRUE).** Значение этого параметра можно устанавливать как на уровне системы, так и на уровне сеанса, а также глобально, в файле параметров инициализации экземпляра. Достаточно просто добавить в файл **INIT.ORA** для экземпляра строку:

```
timed_statistics=true
```

и при следующем перезапуске СУБД этот параметр будет включен. Для установки его на уровне сеанса выполните следующую команду:

```
tkyte@TKYTE816> alter session set timed_statistics=true;

Session altered.
```

А для включения учета времени во всей системе:

```
tkyte@TKYTE816> alter system set timed_statistics=true;

System altered.
```

Как уже упоминалось, получаемая информация настолько ценна, что я собираю ее всегда, установив параметру значение **TRUE** в файле параметров инициализации. Дополнительные расходы ресурсов при этом минимальны, а при отсутствии соответствующей информации о контроле производительности не может быть и речи.

Организация трассировки

Параметр **SQL_TRACE** также можно устанавливать на уровне системы или сеанса. При его установке генерируется так много данных и работа системы так замедляется, что включать ею лучше избирательно (редко или вообще никогда его не устанавливают для системы в файле **init.ora).** Параметр **SQL_TRACE** тоже может иметь одно из двух значений, **TRUE** или FALSE. Если установлено значение TRUE, в каталоге, задаваемом параметром **USER_DUMP_DEST** файла **init.ora** при подключении к выделенному серверу или **BACKGROUND_DUMP_DEST** — при подключении к многопотоковому (MTS) серверу, будут генерироваться трассировочные файлы. Я, однако, не рекомендую использовать **SQL_TRACE** при подключении в режиме MTS, поскольку результаты запросов сеанса будут записываться в различные трассировочные файлы при пере-

ходе сеанса с одного разделяемого сервера на другой. При подключении в режиме MTS интерпретация результатов **SQL_TRACE** практически невозможна. Еще один важный параметр в файле **init.ora — MAX_DUMP_FILE_SIZE.** Он ограничивает максимальный размер генерируемого сервером трассировочного файла. Если обнаружится, что трассировочные файлы — усеченные, увеличьте значение этого параметра. Это можно сделать с помощью оператора **ALTER SYSTEM** или **ALTER SESSION.** Параметр **MAX_DUMP_FILE_SIZE** можно задать тремя способами.

- Числовое значение параметра **MAX_DUMP_FILE_SIZE** задает максимальный размер в блоках файловой системы.
- Число, за которым следует суффикс K или M, задает размер в килобайтах или мегабайтах, соответственно.
- Строка **UNLIMITED** означает, что ограничения на размер трассировочных файлов нет — они могут иметь любой размер, допускаемый операционной системой.

Я не рекомендую устанавливать значение **UNLIMITED** — так можно заполнить всю файловую систему; значения в диапазоне от 50 до 100 Мбайт обычно более чем достаточно.

Как включить параметр **SQL_TRACE?** Чаще всего я использую следующие способы.

- **ALTER SESSION SET SQL_TRACE=TRUE|FALSE.** Выполнение этого оператора SQL позволит включить стандартный режим трассировки **SQL_TRACE** в текущем сеансе. Этот оператор наиболее полезен в интерактивной среде типа SQL*Plus или при встраивании в приложение, так чтобы из приложения можно было при необходимости включать и отключать трассировку. Возможность просто включать и отключать **SQL_TRACE** средствами приложения — будь-то опция командной строки, пункт меню или параметр конфигурации — полезна в любом приложении.
- **SYS.DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION.** Эта процедура позволяет устанавливать и сбрасывать трассировку для любого существующего сеанса. Для этого необходимо указать лишь параметры **SID** и **SERIAL#** соответствующего сеанса — их можно получить из представления динамической производительности **V$SESSION.**
- **ALTER SESSION SET EVENTS.** Можно установить событие, обеспечивающее трассировку с большим объемом регистрируемой информации, чем обычно получается при установке **ALTER SESSION SET SQL_TRACE=TRUE.** Прием с использованием **SET EVENTS** не описан в документации и не поддерживается Oracle Corporation, но его существование общеизвестно (обратитесь на сайт **http://www.google.com/** и поищите по запросу **alter session set events 10046:** вы увидите, на скольких сайтах эта возможность описана). С помощью этого события можно не только получить всю информацию, выдаваемую при установке SQL_TRACE, но и значения связываемых переменных в SQL-операторах, а также информацию о том, какие события ожидаются (что замедляет работу) при выполнении этих SQL-операторов.

Есть и другие методы, но именно эти три чаще всего я встречаю и использую сам. Методы установки **SQL_TRACE** оператором **ALTER SESSION SET SQL_TRACE** и вызовом **SYS.DBMS_SYSTEM** — очень просты и очевидны. Использование события несколько менее тривиально. При этом используется внутренний (и не отраженный в документации) механизм событий сервера Oracle. Если коротко, используются следующие команды:

```
alter session set events '10046 trace name context forever, level <N>';
alter session set events '10046 trace name context off;
```

где N может иметь одно из следующих значений:

N=1. Включает стандартные средства **SQL_TRACE.** Результат не отличается от установки **SQL_TRACE=true.**

N=4. Включает стандартные средства **SQL_TRACE** и добавляет в трассировочный файл значения связываемых переменных.

N=8. Включает стандартные средства **SQL_TRACE** и добавляет в трассировочный файл информацию об ожидании событий на уровне запросов.

N=12. Включает стандартные средства **SQL_TRACE** и добавляет как значения связываемых переменных, так и информацию об ожидании событий.

Использование средств пакета **DBMS_SUPPORT** — еще один метод установки трассировки с отображением значений связываемых переменных и информации об ожиданиях событий. Для получения пакета **DBMS_SUPPORT** надо связаться со службой поддержки Oracle Support, поскольку он не входит в обычно поставляемый набор инструментальных средств. Так как это всего лишь интерфейс к представленной выше команде **ALTER SYSTEM SET EVENTS,** ее использование может оказаться проще.

Теперь, когда известно, как включить **SQL_TRACE,** возникает вопрос, как это средство лучше использовать? Лично я предпочитаю добавлять в свои приложения опцию, которую можно задать в командной строке или в адресе URL (если это Web-приложение), для включения трассировки. Это позволяет получать информацию **SQL_TRACE** для одного сеанса. Многие средства разработки Oracle тоже позволяют это сделать. Например, при использовании Oracle Forms, можно вызывать форму с параметром:

```
C:\> ifrun60 module=myform userid=scott/tiger statistics=yes
```

STATISTICS=YES - это флаг, требующий выполнить команду **ALTER SESSION SET SQL_TRACE=TRUE.** Если бы подобные средства предоставляли все приложения, их настройка упростилась бы. Можно было бы попросить пользователя, столкнувшегося с проблемой производительности, включить трассировку и затем воспроизвести проблему. В результате можно получить всю информацию, необходимую для выяснения причин медленной работы. Не придется спрашивать, что именно делается, чтобы воспроизвести проблему самостоятельно, — достаточно будет попросить ее воспроизвести и затем проанализировать результаты трассировки. Если в трассировочном файле имеются значения связываемых переменных и информация об ожидавшихся событиях, то данных более чем достаточно для выяснения того, что же работает не так.

Как обеспечить трассировку, если приходится работать с приложением стороннего производителя или с существующим приложением, не поддерживающим включение **SQL_TRACE**? Я использую два подхода. Один из них подходит для клиент-серверного приложения, постоянно подключенного к базе данных. Достаточно запустить приложение и подключиться к базе данных. Затем, выполнив запрос к представлению **V$SESSION**, можно определить параметры **SID** и **SERIAL#** соответствующего сеанса. Теперь можно вызвать **SYS.DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION** для включения трассировки указанного сеанса. Сегодня, однако, популярны Web-приложения, для которых этот метод не подходит. Сеансы при этом очень короткие, и они часто начинаются и завершаются. Необходимо установить средство **SQL_TRACE** для пользователя — при каждом подключении этого пользователя необходимо устанавливать SQL_TRACE для соответствующего сеанса. К счастью, это можно сделать с помощью триггера базы данных на событие **LOGON**. Например, в Oracle 8i я часто использую следующий триггер (триггеры на события базы данных, такие как **AFTER LOGON**, — новое средство, поддерживаемое в Oracle 8.1.5 и последующих версиях):

```
create or replace trigger logon_trigger
after logon on database
begin
   if (user = 'TKYTE') then
      execute immediate
      'ALTER SESSION SET EVENTS ''10046 TRACE NAME CONTEXT FOREVER, LEVEL
4 ' " ;
   end if;
end;/
```

Он обеспечивает включение трассировки при каждом подключении к базе данных. Приложение не надо менять для установки SQL_TRACE — мы это сделаем сами.

Использование и интерпретация результатов работы утилиты TKPROF

TKPROF — это средство командной строки для преобразования трассировочного файла в более удобочитаемый вид. Это прекрасная утилита, к сожалению, недостаточно часто используемая. Я считаю, что это связано с тем, что о ее существовании просто не знают. Теперь, когда вы о ней знаете, я думаю, вы будете использовать ее постоянно.

В этом разделе я выполню запрос с включенной трассировкой. Затем мы детально рассмотрим соответствующий отчет TKPROF и выясним, на что надо обращать внимание в подобных отчетах:

```
tkyte@TKYTE816> show parameter timed_statistics;

NAME                          TYPE      VALUE

timed_statistics              boolean   TRUE

tkyte@TKYTE816> alter session set sql_trace=true;
```

```
Session altered.

tkyte@TKYTE816> select owner, count(*)
  2   from all_objects
  3   group by owner;
OWNER                          COUNT(*)
------------------------------ --------
CTXSYS                              185
DBSNMP                                4
DEMO                                  5
DEMO11                                3
MDSYS                               176
MV_USER                               5
ORDPLUGINS                           26
ORDSYS                              206
PUBLIC                             9796
SCOTT                                18
SEAPARK                               3
SYS                                11279
SYSTEM                               51
TKYTE                                32
TYPES                                 3

15 rows selected.

tkyte@TKYTE816> select a.spid
  2     from v$process a, v$session b
  3    where a.addr = b.paddr
  4      and b.audsid = userenv('sessionid')
  5  /

SPID


1124
```

Здесь я проверил, что установлен параметр TIMED_STATISTICS (без этого устанавливать **SQL_TRACE** практически бесполезно), а затем включил **SQL_TRACE.** Затем я выполнил запрос, который необходимо проанализировать. Наконец, я выполнил запрос для получения идентификатора серверного процесса **(SPID** — server process ID) — он потребуется для идентификации соответствующего трассировочного файла. После выполнения этого запроса я завершил сеанс SQL*Plus и перешел в каталог на сервере, заданный в качестве значения параметра **USER_DUMP_DEST** в файле **init.ora.** Значение этого параметра можно получить по ходу сеанса с помощью запроса к представлению **V$PARAMETER** или утилиты **DBMS_UTILITY** (для этого не нужен доступ к представлению **V$PARAMETER):**

```
tkyte@TKYTE816> declare
  2   l_intval number;
  3   l_strval varchar2(2000) ;
  4   l_type    number;
  5   begin
  6       l_type  := dbms_utility.get_parameter_value
  7                                  ('user_dump_dest', l_intval, l_strval);
```

```
  8        dbms_output.put_line(l_strval);
  9    end;
  10   /

C:\oracle\admin\tkyte816\udump

PL/SQL procedure successfully completed.
```

В этом каталоге я обнаружил следующее:

```
C:\oracle\ADMIN\tkyte816\udump>dir
 Volume in drive C has no label.
 Volume Serial Number is F4S5-B3C3

 Directory of C:\oracle\ADMIN\tkyte816\udump

03/16/2001   02:55p              <DIR>
03/16/2001   02:55p              <DIR>
03/16/2001   08:45a                5,114 ORA00860.TRC
03/16/2001   02:52p                3,630 ORA01112.TRC
03/16/2001   02:53p                6,183 ORA01124.TRC
               3 File(s)         14,927 bytes
               2 Dir(s)   13,383,999,488 bytes free
```

Несколько трассировочных файлов — вот теперь и пригодится значение SPID. Трассировочный файл моего сеанса — **ORA01124.TRC.** Я знаю это, потому что значение **SPID** — часть имени файла. В ОС UNIX используется похожий принцип именования, т.е. в имя файла тоже входит значение **SPID.** Одна из проблем, связанных с трассировочными файлами, состоит в том, что они могут быть недоступны для чтения пользователям, не входящим в группу администраторов Oracle (например, в группу **dba** в ОС UNIX). Если они недоступны, попросите администратора базы данных установить параметр

```
_trace_files_public = true
```

в файле **init.ora** на тестовых серверах и серверах разработчиков. Это позволит читать трассировочные файлы на сервере всем пользователям. Эту установку нельзя делать на производственном сервере, поскольку трассировочные файлы могут содержать секретную информацию. В тестовой среде или среде разработки она вполне безопасна. Обратите внимание, что имя параметра начинается с символа подчеркивания. Этот параметр не описан в документации и не поддерживается корпорацией Oracle. Как и в случае с командой EVENTS, которую мы будем использовать чуть позже, этот параметр широко известен и повсеместно используется: поищите с помощью запроса **_trace_files_public** в Google или любой другой поисковой системе и вы получите много интересной информации об этом параметре.

Теперь, определив свой трассировочный файл, необходимо его сформатировать. Можно читать его и непосредственно. Но около 90 процентов нужной информации легко получить из хорошо сформатированного отчета. Остальные 10 процентов информации обычно не нужны, но если она потребуется, придется получать эту информацию непосредственно из трассировочного файла. Для форматирования трассировочного файла ис-

пользуется утилита командной строки TKPROF. В простейшем случае достаточно выполнить:

```
C:\oracle\ADMIN\tkyte816\udump>tkprof ora01124.trc report.txt

TKPROF: Release 8.1.6.0.0 - Production on Fri Mar 16 15:04:33 2001

 (c) Copyright 1999 Oracle Corporation.  All rights reserved.
```

Параметрами команды TKPROF являются имена файла исходных данных и файла результатов обработки. Теперь достаточно открыть файл REPORT.TXT в текстовом редакторе, и мы увидим следующее:

```
select owner, count(*)
from all_objects
group by owner

call        count     cpu   elapsed  disk     query  current     rows
------------------------------------------------------------------------
Parse           1    0.00      0.00     0         0        0        0
Execute         1    0.00      0.00     0         0        0        0
Fetch           2    1.20      1.21     0     86091        4       15
------------------------------------------------------------------------
total                1.20      1.21           86091                15

Misses in library cache during parse: 0
Optimizer goal: CHOOSE
Parsing user id: 69

Rows     Row Source Operation


    15   SORT GROUP BY
 21792    FILTER
 21932     NESTED LOOPS
    46      TABLE ACCESS FULL USER$
 21976      TABLE ACCESS BY INDEX ROWID OBJ$
 21976       INDEX RANGE SCAN (object id 34)
     1   FIXED TABLE FULL X$KZSPR
     1   FIXED TABLE FULL X$KZSPR
     0   FIXED TABLE FULL X$KZSPR
     0   FIXED TABLE FULL X$KZSPR
     1   FIXED TABLE FULL X$KZSPR
     1   FIXED TABLE FULL X$KZSPR
     1   FIXED TABLE FULL X$KZSPR
     1   FIXED TABLE FULL X$KZSPR
     1   FIXED TABLE FULL X$KZSPR
     1   FIXED TABLE FULL X$KZSPR
     1   FIXED TABLE FULL X$KZSPR
     1   FIXED TABLE FULL X$KZSPR
     1   FIXED TABLE FULL X$KZSPR
     1   FIXED TABLE FULL X$KZSPR
 11777   NESTED LOOPS
 30159    FIXED TABLE FULL X$KZSRO
 28971    TABLE ACCESS BY INDEX ROWID OBJAUTH$
```

```
28973     INDEX RANGE SCAN  (object id 101)
  631   TABLE ACCESS BY INDEX ROWID IND$
  654    INDEX UNIQUE SCAN  (object id 36)
```

Утилита TKPROF выдает массу информации. Давайте рассмотрим ее по частям:

```
select owner, count(*)
from all_objects
group by owner
```

Сначала показан исходный запрос в том виде, как он был получен сервером. Так легко можно узнать искомые запросы. В данном случае именно такой запрос я и ввел. Затем идет общая информация о выполнении запроса:

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	1.20	1.21	0	86091	4	15
total	4	1.20	1.21	0	86091	4	15

Здесь можно увидеть три основные стадии выполнения запроса.

- Стадия **PARSE**. На этом этапе сервер Oracle находит запрос в разделяемом пуле (мягкий разбор) или создает новый план его выполнения (жесткий разбор).
- Стадия **EXECUTE**. Это действия, выполняемые сервером Oracle при открытии курсора или выполнении запроса. Для операторов **SELECT** соответствуюшие.столбцы часто будут "пустыми", тогда как для операторов **UPDATE** именно на этой стадии все и делается.
- Стадия **FETCH**. Для оператора **SELECT** именно на этом этапе выполняется основная работа, что и будет отражено в отчете, но для операторов типа **UPDATE** ничего делаться не будет (при выполнении этого оператора данные не извлекаются).

Заголовки столбцов в этом разделе отчета имеют следующие значения:

- **CALL**. Может иметь одно из значений: **PARSE**, **EXECUTE**, **FETCH** или **TOTAL**. Показывает, о какой стадии обработки запроса идет речь.
- **COUNT**. Показывает, сколько раз произошло событие. Это число может оказаться крайне важным. Ниже мы рассмотрим, как интерпретировать значения столбцов.
- **CPU**. Показывает, сколько секунд процессорного времени заняла эта стадия выполнения запроса. Этот столбец заполняется, только если установлен параметр **TIMED_STATISTICS**.
- **ELAPSED**. Показывает, сколько реального времени потребовала эта стадия выполнения запроса. Этот столбец заполняется, только если установлен параметр **TIMED STATISTICS**.

- **DISK.** Показывает, сколько физических операций ввода/вывода с диска потребовалось для выполнения запроса.
- **QUERY.** Показывает, сколько блоков обработал запрос в режиме согласованного чтения. Сюда входят блоки, прочитанные из сегмента отката для получения предыдущего состояния блока.
- **CURRENT.** Показывает, сколько блоков было прочитано в режиме **'CURRENT'.** Блоки в режиме **CURRENT** читаются в том виде, как они есть на момент чтения, а не в режиме согласованного чтения. Обычно блоки для запроса получаются в том виде, как они были на **момент начала** запроса. В текущем режиме блоки извлекаются в том виде, как они существуют на момент их чтения, а не какими они были ранее. В ходе выполнения оператора **SELECT** можно увидеть извлечения в режиме **CURRENT,** связанные с чтением словаря данных в поисках следующего экстента таблицы при полном просмотре (необходима текущая информация об этом, а не согласованное чтение). В ходе изменения мы будем также обращаться к блокам в режиме **CURRENT.**
- **ROWS.** Показывает, сколько строк было затронуто на данной стадии обработки. При выполнении оператора **SELECT строки** будут обрабатываться на стадии **FETCH.** При выполнении оператора **UPDATE** количество обработанных строк будет показано на стадии **EXECUTE.**

В этом разделе отчета надо обратить внимание на следующие особенности.

Значительный процент (около 100) разборов по отношению к выполнениям, если количество выполнений — более одного. Берем количество разборов оператора и делим на количество выполнений. Если получаем в результате 1, значит, запрос разбирался при каждом выполнении, и это надо исправить. Желательно, чтобы это отношение стремилось к нулю. В идеале разбор должен быть один, а выполнений — более одного. Если наблюдается значительное количество разборов, значит, выполняется многократный мягкий разбор запроса. Как было показано в предыдущем разделе, это может существенно снизить масштабируемость и производительность даже единственного пользовательского сеанса. Необходимо обеспечить однократный разбор и многократное выполнение запроса в сеансе; от разбора SQL-оператора при каждом выполнении надо избавляться.

Одно выполнение для всех или почти всех операторов SQL. Если в отчете **TKPROF** указано, что все операторы SQL выполняются только один раз, скорее всего, не используются связываемые переменные (выполняются похожие запросы, которые отличаются лишь используемыми константами). В трассировочных файлах реальных приложений уникальных операторов SQL обычно немного; одни и те же SQL-операторы выполняются многократно. Слишком большое количество уникальных SQL-операторов обычно означает, что недостаточно используются связываемые переменные.

Существенное отличие процессорного и реального времени выполнения запроса. Это означает, что приходится долго чего-то ждать. Если для выполнения необходима одна секунда процессорного времени, но реально запрос выполнялся 10 секунд, это означает, что 90 процентов времени ушло на ожидание освобождения ресурса. Далее в этом разделе мы увидим, как по исходному файлу трассировки определить причину ожида-

ния. Это ожидание может быть вызвано несколькими причинами. Например, на выполнение изменения, заблокированного другим сеансом, уйдет намного больше реального времени, чем процессорного. SQL-запрос, выполняющий большой объем физического ввода/вывода с диска, может долго ждать завершения ввода/вывода.

Длительное процессорное или реальное время выполнения. Сокращение продолжительности длительно выполняющихся запросов — ваша ближайшая.цель. Если удастся ускорить их выполнение, программа заработает быстрее. Зачастую, один запрос-монстр тормозит всю работу; настройте его, и приложение будет отлично работать.

Большая величина отношения (FETCH COUNT)/(количество извлеченных строк). Для ее вычисления берем количество действий типа **FETCH** (в нашем примере — два) и делим на количество извлеченных строк (в нашем примере — 15). Если полученный результат близок к одному и извлечено более одной строки, приложение не выполняет множественные извлечения. Любой язык или функциональный интерфейс позволяет это делать — извлекать несколько строк одним вызовом. Если возможность множественного извлечения не используется, на пересылки информации с клиента на сервер и обратно уйдет намного больше времени. Этот постоянный обмен информацией, помимо того, что чрезвычайно загружает сеть, выполняется намного медленнее, чем получение нескольких строк одним вызовом. Как организовать множественное извлечение данных в приложении, зависит от используемого языка и/или функционального интерфейса. Например, при использовании Pro*C необходимо выполнять перекомпиляцию с параметром **prefetch=NN,** в Java/JDBC необходимо вызвать метод **SETROWPREFETCH,** в **PL/**SQL необходимо использовать конструкцию **BULK COLLECT** в операторах **SELECT INTO** и т.д. Представленный ранее пример показывает, что утилита SQL*Plus (использованный нами клиент) вызвала **FETCH** дважды для извлечения 15 строк. Это показывает, что утилита SQL*Plus использовала при выборке массив не менее чем из восьми строк. На самом деле стандартным для SQL*Plus является использование массивов из 15 строк; вторая операция извлечения вернула ноль строк — она получила признак конца результирующего множества.

Слишком большое количество физических обращений к диску. Простое правило для этого параметра придумать сложнее, но если **DISK COUNT = QUERY + CURRENT MODE BLOCK COUNT,** значит, все блоки читались с диска. Будем надеяться, что при следующем выполнении этого запроса часть блока будет найдена в области SGA. Большое количество обращений к диску — предупреждающий сигнал о том, что необходимо провести дополнительное исследование. Возможно, надо увеличить размер буферного кэша в SGA или придумать другой запрос, требующий чтения меньшего количества блоков.

Слишком большое количество обработанных блоков (QUERY или **CURRENT).** Это показывает, что запрос обрабатывает большой объем информации. Проблема это или нет — судить вам. Некоторым запросам действительно необходимо обработать много данных, как в представленном ранее примере. Часто выполняемый запрос, однако, должен обрабатывать сравнительно немного блоков. Если сложить значения **QUERY** и **CURRENT** количества обработанных блоков и поделить на значение столбца **count** в строке **EXECUTE,** должно получаться небольшое число.

Давайте перейдем к следующей части отчета:

```
Misses in library cache during parse: 0
Optimizer goal: CHOOSE
Parsing user id: 69
```

Из этого можно сделать вывод, что выполненный запрос был найден в разделяемом пуле (количество не найденных в библиотечном кэше запросов равно 0). То есть, выполнялся мягкий разбор запроса. При самом первом выполнении запроса в этой строке будет значение 1. Если практически у всех выполнявшихся запросов будет значение 1, значит, не используются связываемые переменные (это надо исправить). Операторы SQL не используются повторно.

Вторая строка показывает режим работы оптимизатора при выполнении запроса. Эта информация — для справки; выбранный и использованный план выполнения запроса зависит от этого режима.

Наконец, представлен идентификатор пользователя, разобравшего запрос. По этому идентификатору можно получить имя пользователя:

```
tkyte@TKYTE816> select * from all_users where user_id = 69;

USERNAME              USER ID CREATED
--------------  ------------- -----------------
TKYTE                      69 10-MAR-01
```

Как видите, запрос разбирал я. Последний раздел отчета TKPROF для данного запроса — план выполнения. Стандартный план выполнения показан ниже:

```
Rows      Row Source Operation

      15  SORT GROUP BY
   21792   FILTER
   21932    NESTED LOOPS
      46     TABLE ACCESS FULL USER$
   21976     TABLE ACCESS BY INDEX ROWID OBJ$
   21976      INDEX RANGE SCAN (object id 34)
       1    FIXED TABLE FULL X$KZSPR
       1    FIXED TABLE FULL X$KZSPR
       0    FIXED TABLE FULL X$KZSPR
       0    FIXED TABLE FULL X$KZSPR
       1    FIXED TABLE FULL X$KZSPR
       1    FIXED TABLE FULL X$KZSPR
       1    FIXED TABLE FULL X$KZSPR
       1    FIXED TABLE FULL X$KZSPR
       1    FIXED TABLE FULL X$KZSPR
       1    FIXED TABLE FULL X$KZSPR
       1    FIXED TABLE FULL XSKZSPR
       1    FIXED TABLE FULL X$KZSPR
       1    FIXED TABLE FULL X$KZSPR
       1    FIXED TABLE FULL X$KZSPR
   11777    NESTED LOOPS
   30159     FIXED TABLE FULL X$KZSRO
   28971     TABLE ACCESS BY INDEX ROWID OBJAUTH$
```

```
28973        INDEX RANGE SCAN  (object id 101)
  631      TABLE ACCESS BY INDEX ROWID  IND$
  654        INDEX UNIQUE SCAN  (object id 36)
```

Это реальный план запроса, использованный сервером Oracle при выполнении. Интересно, что показано количество обработанных на каждом шаге строк. Можно увидеть, например, что 28971 строка была извлечена из **OBJAUTH$.** Речь идет о строках, полученных в результате выполнения данного шага плана (после применения всех возможных условий, которым должны соответствовать строки таблицы **OBJAUTH$,** на следующий шаг плана передана 28971 строка). В Oracle 8.0 и более ранних версиях выдавалось количество строк, просмотренных на данном шаге плана выполнения (количество строк, поступивших на вход этого шага). Например, если рассматривалось 50000 строк в таблице **OBJAUTH$,** но часть из них была исключена конструкцией **WHERE,** в отчете **TKPROF** версии Oracle 8.0 выдано было бы в соответствующей строке плана 50000. По этой информации можно понять, от каких шагов выполнения запроса имеет смысл отказаться либо путем изменения запроса, либо с помощью подсказок оптимизатору, выбирающих более удачный план.

Обратите внимание, что вперемешку используются как имена объектов (например, **TABLE ACCESS BY INDEX ROWID INDS),** так и идентификаторы (например, **INDEX UNIQUE SCAN (object id 36)).** Причина в том, что в исходном трассировочном файле для некоторых объектов не записаны имена, а только идентификаторы. Кроме того, по умолчанию утилита **TKPROF** не подключается к базе данных для преобразования идентификаторов объектов в имена. Получить имя объекта по идентификатору можно с помощью запроса:

```
tkyte@TKYTE816> select owner, object_type, object_name
  2   from all_objects
  3   where object_id = 36;

OWNER                OBJECT_TYPE              OBJECT_NAME

SYS                          INDEX              I_IND1
```

Но можно и добавить параметр **EXPLAIN=** при вызове утилиты **TKPROF** следующим образом:

```
C: \oracle\ADMIN\tkyte816\uduinp>tkprof ora01124.trc x.txt explain=tkyte/tkyte
```

В результирующем файле мы получим следующее сообщение об ошибке:

```
error during parse of EXPLAIN PLAN statement
ORA-01039: insufficient privileges on underlying objects of the view
```

Хотя мы и можем выполнить запрос, базовые таблицы, по которым построено представление, недоступны. Чтобы получить план выполнения запроса, необходимо подключиться от имени учетной записи SYS или другой учетной записи, имеющей доступ к базовым объектам.

*Я предпочитаю не использовать параметр **EXPLAIN=** и вам не советую.*

Запрос, передаваемый оператору **EXPLAIN PLAN,** может принципиально отличаться от используемого при реальном выполнении. Единственный план, которому можно доверять, — это план, сохраненный в самом трассировочном файле. Вот простой пример использования утилиты **TKPROF** с параметром **explain=имя/пароль,** демонстрирующий это:

```
select  count(object_type)
from
 t where object_id > 0

call      count       cpu    elapsed  disk    query    current

Parse         1      0.00       0.00      0        0          0        0
Execute       1      0.00       0.00      0        0          0        0
Fetch         2      0.19       2.07    337    20498          0        1

total                0.19       2.07    337    20498

Misses in library cache during parse:  1
Optimizer goal:  CHOOSE
Parsing user id:  69   (TKYTE)

Rows       Row Source Operation
--------   ---------------------------------------------------
           1   SORT AGGREGATE
   21790    TABLE ACCESS BY INDEX ROWID T
   21791      INDEX RANGE SCAN  (object id 25291)


Rows       Execution Plan
--------   ---------------------------------------------------
           0   SELECT  STATEMENT    GOAL:  CHOOSE
           1     SORT  (AGGREGATE)
   21790     TABLE ACCESS    GOAL:  ANALYZED  (FULL)  OF  'T'
```

Очевидно, один из планов — неправильный; в одном указано сканирование диапазона по индексу и доступ к таблице по идентификатору строки, а в другом —полный просмотр. Если не знать, ***что я проанализировал таблицу после выполнения запроса, но до запуска утилиты*** **TKPROF,** это расхождение объяснить нельзя. После анализа таблицы стандартный план выполнения этого запроса изменился. Утилита **TKPROF** использует предоставляемый Oracle стандартный оператор **explain plan.** В результате выдается план, который использовался бы при выполнении оператора сейчас, а **не** использованный фактически. Различие планов в трассировочном файле и в результатах выполнения оператора **explain plan** может обуславливаться многими факторами. Например, приложение могло использовать хранимые шаблоны запросов (подробнее об этой возможности см. в главе 11). В ходе выполнения план мог базироваться на хранящейся схеме, а план, возвращенный оператором **explain plan,** может оказаться другим. В общем случае, если при вызове утилиты **TKPROF** все же используется параметр **EXPLAIN=,** необходимо пошагово сравнить согласованность двух полученных планов.

Утилита **TKPROF** имеет много опций командной строки, и если ввести команду **tkprof,** все они будут выданы:

```
C:\Documents and Settings\Thomas Kyte\Desktop>tkprof

Usage: tkprof tracefile outputfile [explains ] [tables ]
                  [prints ] [insert= ] [sys= ] [sorts ]
  table=имя_схемы.имя таблицы  Используйте "имя_схемы.имя_таблицы" внесте с
                               опцией "explains".
  explain=пользователь/пароль  Подключиться к ORACLE и выполнить EXPLAIN PLAIN.
  print=количество    Выдать только указанное "количество" операторов SQL.
  aggregate=yes|no
  insert=имя_файла    Выдать в этот файл операторы SQL и данные в операторах
                      INSERT.
  sys=no              Не выдавать информацию об операторах SQL, выполненных от
                      имени пользователя SYS.
  record=имя_файла    Выдать сюда нерекурсивные операторы, имеющиеся в
                      трассировочном файле.
  sort=опции          Набор из куля или более следующих опций:
    prscnt  сколько раз выполнялся разбор
    prscpu  процессорное время раэбора
    prsela  реальное время раэбора
    prsdsk  количество чтений с диска в ходе разбора
    prsqry  количество буферов, прочитанных в режиме согласованного чтения в
            ходе разбора
    prscu   количество буферов, непосредственно прочитанных в ходе раэбора
    prsmis  количество непопаданий в библиотечный кэше в ходе разбора
    execnt  сколько раз выполнялся оператор
    execpu  процессорное время выполнения
    exeela  реальное время выполнения
    exedsk  количество чтений с диска при выполнении
    exeqry  количество буферов, прочитанных в режиме согласованного чтения в
            ходе выполнения
    execu   количество буферов, непосредственно прочитанных при выполнении
    exerow  количество обработанных при выполнении строк
    exemis  количество непопаданий в библиотечный кэш в ходе выполнения
    fchcnt  сколько раз выполнялось извлечение данных
    fchcpu  процессорное время извлечения данных
    fchela  реальное время извлечения данных
    fchdsk  количество обращений к диску при извлечении данных
    fchqry  количество буферов, прочитанных в режиме согласованного чтения при
            извлечении данных
    fchcu   количество буферов, непосредственно прочитанных при извлечении
            данных
    fchrow  количество извлеченных строк
    userid  идентификатор пользователя, разобравшего оператор
```

Наиболее полезной, по моему мнению, является опция sort=. Я люблю сортировать результаты по разным показателям процессорного и реального времени выполнения, чтобы "наихудшие" запросы оказывались в начале трассировочного файла. Сортировку можно также использовать для поиска запросов, выполняющих слишком много физическо-

Описания параметров переведены на русский язык. Для получения исходных описаний на английском выполните команду tkprof. *Прим. научн. ред.*

го ввода/вывода и т.д. Назначение остальных опций очевидно. В 99,9 процентах случаев я использую **tkprof имя_трассировочного_файла имя_файла_отчета,** и ничего более. При этом операторы SQL выдаются примерно в том порядке, как они посылались серверу в ходе выполнения. Я могу использовать утилиту типа **grep** в ОС UNIX или **find** в Windows для извлечения суммарных **(total)** строк, что позволяет легко определить, на какие запросы надо обратить внимание. Например, обработав полученный ранее файл **report.txt:**

```
C:\oracle\ADMIN\tkyte816\udump>find "total" report.txt

REPORT.TXT
total    2     0.00       0.00       0          0       0         0
total    4     0.01       0.02       1          1       4         1
total    4     1.20       1.21       0      86091       4        15
total    6     0.01       0.01       0          4       0         2
total    4     0.00       0.00       0          0       0         1
total   14     1.21       1.23       1      86092       8        17
total    6     0.01       0.01       0          4       0         2
```

можно понять, что для ускорения процесса надо в текстовом редакторе искать строку 1.21. Есть и другие операторы, но, очевидно, именно на этом надо сконцентрировать усилия для ускорения работы приложения.

Использование и интерпретация исходных трассировочных файлов

В СУБД Oracle есть два типа трассировочных файлов: генерируемые при установке SQL_TRACE (их мы и рассматриваем) и генерируемые при сбое сеанса (в результате ошибки в СУБД). Трассировочные файлы второго типа, получаемые при сбое сеанса, непосредственно разработчикам не нужны — их посылают в службу поддержки Oracle Support для анализа. Трассировочные файлы первого типа разработчикам **очень** нужны, особенно если знать, как их читать и интерпретировать.

В большинстве случаев трассировочные файлы обрабатываются и форматируются утилитой **TKPROF,** но периодически необходимо изучать исходный трассировочный файл, чтобы получить больше информации о происходящем, чем выдает **TKPROF.** Допустим, имеется отчет **TKPROF** со следующей информацией:

```
UPDATE BMP SET ENAME=LOWER(ENAME)
WHERE
 EMPNO = :b1


call      count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse         1      0.00       0.00          0          0          0           0
Execute       1      0.00      54.25          0         17          8           1
Fetch         0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total         2      0.00      54.25          0         17
```

Проблема, очевидно, есть: на изменение одной строки уходит почти минута, хотя процессорного времени требуется менее сотой доли секунды. Итак, слишком долго пришлось ожидать какого-то события, но какого именно, утилита TKPROF не показывает. Кроме того, не мешало бы знать, при изменении какой строки это произошло (а именно: какое значение EMPNO было указано в переменной :b1). Эта информация поможет понять, как мы попали в подобную ситуацию. К счастью, трассировка приложения обеспечивалась следующей командой:

```
alter session set events '10046 trace name context forever, level 12';
```

так что в трассировочном файле есть как информация об ожидаемых событиях, так и значения связываемых переменных. Давайте рассмотрим исходный трассировочный файл, от начала до конца. Я трассировал следующий фрагмент кода:

```
scott@TKYTE816> alter session set events
  2  '10046 trace name context forever, level 12';

Session altered.

scott@TKYTE816> declare
  2  l_empno number default 7698;
  3  begin
  4      update emp set ename = lower(ename) where empno = l_empno;
  5  end;
  6  /

PL/SQL procedure successfully completed.

scott@TKYTE816> exit
```

В этом случае мы точно знаем, какое значение EMPNO использовалось, но обычно это не известно. Ниже представлено содержимое трассировочного файла и соответствующие комментарии:

```
Dump file C:\oracle\admin\tkyte816\udump\ORA01156.TRC
Sat Mar 17 12:16:38 2001
ORACLE V8.1.6.0.0 - Production vsnsta=0
vsnsql=e vsnxtr=3
Windows 2000 Version 5.0 , CPU type 586
Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option
JServer Release 8.1.6.0.0 - Production
Windows 2000 Version 5.0 , CPU type 586
Instance name: tkyte816

Redo thread mounted by this instance: 1

Oracle process number: 11

Windows thread id: 1156, image: ORACLE.EXE
```

Это стандартный заголовок трассировочного файла. Он пригодится для определения точной версии системы и СУБД, с которой работает приложение. В заголовке также имеется идентификатор Oracle SID (имя экземпляра), который позволяет понять, с тем ли трассировочным файлом мы работаем.

```
*** 2001-03-17 12:16:38.407
*** SESSION ID:(7.74) 2001-03-17 12:16:38.407
APPNAME mod='SQL*PLUS' mh=3669949024 act='' ah=4029777240
```

Запись **APPNAME** была сделана при вызове подпрограммы пакета **DBMS_APPLICATION_INFO** (подробнее об этом пакете см. в Приложении А). Этот пакет используется для регистрации действий приложений в базе данных, чтобы по результатам запросов к представлению **V$SESSION** можно было понять, какое именно приложение открыло сеанс. Утилита SQL*Plus, в частности, этот пакет использует. В вашем трассировочном файле записи APPNAME может и не быть: все зависит от среды. Было бы **замечательно,** если бы все приложения регистрировались с помощью этого пакета, так что, надеюсь, вы обнаружили эту запись, в которой указано имя работающего модуля. Эта запись имеет следующий формат:

APPNAME mod='%s' mh=%lu act='%s' ah=%lu

Поле	***Значение***
mod	Имя модуля, переданное DBMS_APPLICATION_INFO
mh	Хэш-значение для модуля
act	Действие модуля, переданное DBMS_APPLICATION_INFO
ah	Хэш-значение для действия

Если вы программируете на языке С, то узнаете строку формата для функции **printf** стандартной библиотеки С. По ней можно определить, какого типа данные будут в записи **APPNAME; %s** — это строка, %lu — длинное целое без знака (число). Далее в моем трассировочном файле идут строки:

```
PARSING IN CURSOR #3 len=70 dep=0 uid=54 oct=42 lid=54 tim=6184206
hv=347037164
   ad='31883a4'
alter session set events '10046 trace name context forever, level 12'
END OF STMT
EXEC #3:c=0,e=0,p=0,cr=0,cu=0,mis=l,r=0,dep=0,og=4,tim=6184206
WAIT #3: nam='SQL*Net message to client' ela= 0 pl=1111838976 p2=l p3=0
WAIT #3: nam='SQL*Net message from client' ela= 818 pl-1111838976 p2=l p3=0
```

Здесь можно увидеть, каким именно оператором включена трассировка. Перед этим идет запись **CURSOR,** которая будет в трассировочном файле всегда (все SQL-операторы в трассировочном файле предваряются записью **CURSOR).** Эта запись имеет следующий формат:

Parsing in Cursor #%d len=%d dep=%d uid=%ld oct=%d lid=%ld tim=%ld hv=%ld ad='%s'

Поле	*Значение*
Cursor #	Номер курсора. Его можно использовать, в частности, для определения максимального количества открытых курсоров в приложении, поскольку это значение увеличивается и уменьшается на единицу при каждом открытии нового и закрытии существующего курсора, соответственно.
len	Длина соответствующего SQL-оператора.
dep	Рекурсивная глубина SQL-оператора. Рекурсивный SQL-оператор - это SQL-оператор, инициированный другим SQL-оператором. Обычно рекурсивные SQL-операторы выполняются сервером Oracle для разбора запроса или управления пространством. Это могут быть также SQL-операторы, вызванные в программных единицах PL/SQL (программная единица или блок PL/SQL - это тоже SQL). Так что операторы приложения тоже могут оказаться "рекурсивными".
uid	Идентификатор пользователя - владельца текущей схемы. Обратите внимание, что значение может отличаться от представленного ниже идентификатора **lid,** в частности, если использовался оператор **alter session set current_schema** для изменения схемы, в которой выполняется разбор.
oct	Oracle Command Type. Числовой код, показывающий тип выполняемого оператора SQL.
lid	Идентификатор пользователя, от имени которого выполнялась проверка привилегий доступа.
tim	Таймер с точностью до сотых долей секунды. Сравнивая время регистрации событий, можно определить, насколько отдалены они друг от друга по времени.
hv	Хэш-идентификатор SQL-оператора.
ad	Значение в столбце **ADDR** строки представления **V$SQLAREA,** описывающей SQL-оператор.

Затем в трассировочном файле можно увидеть, что оператор выполнялся сразу же после разбора. Запись **EXEC** имеет следующий формат:

EXEC Cursor#: c=%d, e=%d,p=%d, cr=%d, cu=%d,mi.s=%d, r=%d, dep=%d, og=%d, tim=%d

Поле	*Значение*
Cursor #	Номер курсора,
c	Процессорное время выполнения в сотых долях секунды,
e	Реальное время выполнения в сотых долях секунды,
p	Количество выполненных физических чтений.
cr	Количество чтений блоков в согласованном режиме (логический ввод/вывод).
cu	Количество непосредственных чтений блоков (логический ввод/вывод).

Поле	*Значение*
mis	Количество непопаданий в библиотечный кэш, по которому можно судить, что пришлось разбирать оператор, поскольку он был удален из разделяемого пула как устаревший, никогда не был в разделяемом пуле или версию в кэше вообще нельзя использовать.
r	Количество обработанных строк.
dep	Рекурсивная глубина SQL-оператора.
og	Цель оптимизации, 1= все строки, 2 = первые строки, 3 = оптимизация на основе правил, 4 = выбор режима оптимизации
tim	Таймер.

Есть и другие разновидности записи EXEC, с другими ключевыми словами вместо EXEC:

Поле	*Значение*
PARSE	Разбор оператора.
FETCH	Извлечение строк из результирующего множества курсора.
UNMAP	Освобождение временных сегментов с промежуточными результатами, когда эти результаты уже не нужны.
SORT UNMAP	То же, что и **UNMAP,** но для сегментов сортировки.

Записи **PARSE, FETCH, UNMAP** и **SORT UNMAP** содержат ту же информацию, что и запись **EXEC,** причем, в том же порядке.

Последняя часть этого раздела содержит первые упоминания об ожидании событий. В данном случае это:

```
WAIT #3: nam='SQL*Net message to client' ela= 0 p1=1111838976 p2=1 p3=0
WAIT #3: nam='SQL*Net message from client' ela= 818 p1=1111838976 p2=1 p3=0
```

Это типичные ожидания данных при пересылке с клиента на сервер, которые уже описывались ранее в этой главе. Строка, содержащая **message to client,** означает, что сервер послал клиенту сообщение и ждет ответа. Строка, содержащая **message from client,** означает, что сервер ждет запроса от клиента. В данном случае реальное время ожидания **(ela)** этого события составило 8,18 секунды. Это означает, что я подождал 8,18 секунды после выполнения оператора **ALTER SESSION,** прежде чем послать следующую команду данного примера. Если только не обрабатывается постоянный и непрерывный поток обращений к серверу, ожидание **"message from client"** неизбежно и вполне нормально. Запись **WAIT** имеет следующий формат:

```
WAIT Cursor#: nam='%s' ela=%d p1=%ul p2=%ul p3=%ul
```

Поле	*Значение*
Cursor #	Номер курсора.
nam	Имя ожидаемого события. В руководстве *Oracle Server Reference* содержится полный список событий, которые может ожидать сервер, с подробным описанием каждого события.

Поле	*Значение*
ela	Реальное время ожидания события в сотых долях секунды.
p1, p2, p3	Параметры ожидаемого события. Каждое событие имеет определенный набор параметров. Значение параметров p1, p2 и p3 для конкретного события см. в руководстве *Oracle Server Reference*

Теперь можно переходить к первому реальному оператору в трассировочном файле:

```
PARSING IN CURSOR #3 len=110 dep=0 uid=54 oct=47 lid-54 tim=6185026
hv=2018962105
  ad='31991c8'
declare
l_empno number default 7698;
begin
   update emp set ename = lower(ename) where empno = l_empno;
end;
END OF STMT
PARSE #3:c=0,e=0,p=0,cr=0,cu=O,mis=l,r=0,dep=O,og=4,tim=6185026
BINDS #3:
```

Мы видим блок кода PL/SQL в том виде, в каком мы его передали. По записи PARSE можно понять, что разобран он был очень быстро, хотя и не был найден в библиотечном кэше (MIS=1). Далее идет запись BINDS без детальной информации, поскольку в переданном блоке кода не использовались связываемые переменные. В дальнейшем мы еще вернемся к этой записи. Переходим к следующему оператору в трассировочном файле, где уже можно обнаружить кое-что интересное:

```
PARSING IN CURSOR #4 len-51 dep=l uid=54 oct=6 lid=54 tim=6185026
hv=2518517322
  ad='318e29c'UPDATE EMP SET ENAME=LOWER(ENAME) WHERE EMPNO = :b1
END OF STMT
PARSE#4:c=0,e=0,p=0,cr=0,cu=O,mis=l,r=0,dep=l,og=0,tim=6185026
BINDS #4:
 bind 0: dty=2 mxl=22(21) mal=00 scl=00 pre=00 oacflg=03 oacfl2=l size=24
offset=0
   bfp=07425360 bln=22 avl=03 flg=05
   value=7698
WAIT #4: nam='enqueue' ela= 308 pl=1415053318 p2-393290 p3=2947
WAIT #4: nam='enqueue' ela= 307 pl=1415053318 p2=393290 p3=2947
WAIT #4: nam='enqueue' ela= 307 pl=1415053318 p2=393290 p3=2947
WAIT #4: nam='enqueue' ela= 308 pl=1415053318 p2=393290 p3=2947
WAIT #4: nam='enqueue' ela= 307 pl=1415053318 p2=393290 p3=2947
WAIT #4: nam='enqueue' ela= 308 pl=1415053318 p2=393290 p3=2947
WAIT #4: nam='enqueue' ela= 307 pl=1415053318 p2=393290 p3=2947
WAIT #4: nam='enqueue' ela= 308 pl=1415053318 p2=393290 p3=2947
WAIT #4: nam='enqueue' ela= 307 pl=1415053318 p2=393290 p3=2947
WAIT #4: nam='enqueue' ela= 308 pl=1415053318 p2=393290 p3=2947
WAIT #4: nam='enqueue' ela= 307 pl=1415053318 p2=393290 p3=2947
WAIT #4: nam='enqueue' ela= 307 pl=1415053318 p2=393290 p3=2947
```

```
WAIT #4: nam='enqueue' ela= 308 pl=1415053318 p2=393290 p3=2947
WAIT #4: nam='enqueue' ela= 307 p1=1415053318 p2=393290 p3=2947
WAIT #4: nam='enqueue' ela= 308 pl=1415053318 p2=393290 p3=2947
WAIT #4: nam='enqueue' ela= 307 pl=1415053318 p2=393290 p3=2947
WAIT #4: nam='enqueue' ela= 308 pl=1415053318 p2=393290 p3=2947
WAIT #4: nam='enqueue' ela= 198 pl-1415053318 p2=393290 p3=2947
EXEC #4:c=0,e=5425,p=0,cr=17,cu=8,mis=0,r=l,dep=l,og=4,tim=6190451
EXEC #3:c=0,e=5425,p=0,cr=17,cu=8,mis=0,r=l,dep=0,og=4,tim=6190451
WAIT #3: nam='SQL*Net message to client' ela= 0 pl=1111838976 p2=l p3=0
WAIT #3: nam='SQL*Net message from client' ela= 0 pl=1111838976 p2=l p3=0
```

Представлен оператор **UPDATE** в том виде, как его получил сервер Oracle. Он отличается от того, который использовался в PL/SQL-блоке, а именно: ссылка на **l_empno** (переменная) заменена связываемой переменной. Перед выполнением SQL-оператора PL/SQL-машина заменяет в нем все вхождения локальных переменных связываемыми переменными. Кроме того, по записи **PARSING IN CURSOR** можно понять, что рекурсивная глубина **(dep)** теперь — 1, а не 0, как у исходного блока PL/SQL. Понятно, что это SQL-оператор, выполненный каким-то **другим** SQL- или PL/SQL-оператором; он не передавался клиентским приложением на сервер. Этот флаг можно использовать для поиска соответствующего SQL-оператора. Если поле **dep** имеет ненулевое значение, значит, выполнение SQL-оператора инициировано сервером, а не клиентским приложением. Это можно учесть при настройке. SQL-оператор, выполнение которого инициировано сервером, **легко** изменить без изменения приложения. Для изменения SQL-оператора, посылаемого клиентским приложением, необходимо найти соответствующее приложение, изменить код, перекомпилировать и снова установить его. Чем больше SQL-операторов находится на сервере, тем лучше — их можно изменить, не изменяя само приложение.

Для этого оператора тоже выдана запись **BINDS,** причем с детальной информацией. Наш оператор изменения содержит связываемую переменную, и можно явно узнать ее значение — первая связываемая переменная имела значение 7698. Теперь, если бы этот запрос был проблемным (выполнялся бы медленно), то имеется вся необходимая для его настройки информация. Есть точный текст запроса. Известны значения связываемых переменных (так что можно повторно выполнить его с теми же данными). Известно даже, ожидание каких событий замедлило выполнение. Не хватает только плана выполнения запроса, но это лишь потому, что мы до него еще не дошли.

Запись **BIND** в трассировочном файле содержит следующую информацию:

Поле	***Значение***
cursor #	Номер курсора.
bind N	Позиция связываемой переменной (0 - первая связываемая переменная).
dty	Тип данных (см. ниже).
mxl	Максимальная длина связываемой переменной.
mal	Максимальная длина массива (при связывании массивов или множественных операциях).

Поле	*Значение*
scl	Масштаб.
pre	Точность.
oacflg	Внутренние флаги. Если это число - нечетное, связываемая переменная может иметь пустое значение (допускается значение **NULL).**
oacfl2	Продолжение внутренних флагов.
size	Размер буфера.
offset	Используется при частичных связываниях.
bfp	Адрес связывания.
bin	Длина буфера связывания.
avl	Длина фактического значения.
flag	Внутренние флаги.
value	Представление связываемого значения в виде строки (может быть шестнадцатиричным представлением двоичных данных) - именно это нам и надо!

Значение **dty** (тип данных) можно декодировать с помощью информации из представления **USER_TAB_COLUMNS.** Если выбрать из представления **all_views** текст представления, для которого **view_name = 'USER_VIEWS',** можно увидеть функцию декодирования, сопоставляющую значениям **dty** строковые названия типов.

Интересующая нас информация — информация об ожиданиях — найдена. Можно четко увидеть, почему выполнение изменения потребовало почти минуту реального времени, хотя процессорное время, необходимое для этого, пренебрежимо мало. Мы ожидали снятия блокировки: в главах 3 и 4 было описано, что **enqueue** — один из двух внутренних механизмов, используемых сервером Oracle для поочередного доступа к разделяемым ресурсам. Трассировочный файл показывает, что мы ждали снятия блокировки, т.е. мы не ждали завершения ввода/вывода, синхронизации журнального файла или освобождения буфера, — мы стояли в очереди в ожидании освобождения некоторого ресурса. Если пойти дальше, можно взять значение параметра **p1** и получить по нему тип блокировки, снятия которой пришлось ждать. Вот как это можно сделать:

```
tkyte@TKYTE816> create or replace
  2  function enqueue_decode(l_p1 in number) return varchar2
  3  as
  4      l_str varchar2(25);
  5  begin
  6      select chr(bitand(l_p1,-16777216)/16777215) ||
  7             chr(bitand(l_p1, 16711680)/65535)  ||  '  ' ||
  8             decode(bitand(l_p1, 65535),
  9                    0, 'No lock',
 10                    1, 'No lock',
 11                    2, 'Row-Share',
 12                    3, 'Row-Exclusive',
 13                    4, 'Share',
```

```
 14                              5,  'Share Row-Excl',
 15                              6,  'Exclusive'  )
 16          into l_str
 17          from dual;
 18
 19        return l_str;
 20   end;
 21   /

Function created.

tkyte@TKYTE816>
tkyte@TKYTE816> select enqueue_decode(1415053318) from dual;

ENQUEUE_DECODE(1415053318)


TX Exclusive
```

Этот результат показывает, что мы ждем снятия исключительной блокировки строки. Теперь понятно, почему для изменения строки потребовалась минута. Другой сеанс целую минуту блокировал соответствующую строку, пока мы ждали снятия блокировки. Что делать в этом случае — зависит от приложения. Например, в рассмотренном выше случае я делал изменение "вслепую". Если не хочется, чтобы приложение блокировалось при изменении, можно выполнять его так:

```
select ename from emp where empno = :bv for update NOWAIT;
update emp set ename = lower(ename) where empno = :bv;
```

Это решает проблему блокирования. По крайней мере теперь точно известно, почему изменение выполнялось так долго. Мы можем постфактум это определить. Больше не нужно заниматься диагностикой "на месте" — достаточно получить соответствующую трассировочную информацию.

Ближе к концу трассировочного файла мы видим:

```
PARSING IN CURSOR #5 len=52 dep=0 uid=54 oct=47 lid=54 tim=6190451
   hv=1697159799 ad='3532750'
BEGIN DBMS_OUTPUT.GET_LINES(:LINES, :NUMLINES) ; END;
END OF STMT
PARSE #5:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=6190451
BINDS #5:
 bind 0: dty=1 mxl=2000(255) mal=25 scl=00 pre=00 oacflg=43 oacf12=10
   size*2000
   offset=0
    bfp=07448dd4 Вп=255 avl-00 flg=05
 bind 1: dty=2 mxl=22(02) mal=00 scl=00 pre=00 oacflg=01 oacf12=0 size=24
   offset=0
    bfp=0741c7e8 bln=22 avl=02 flg=05
    value=25
WAIT #5: nam='SQL*Net message to client' ela= 0 pl=1111838976 p2=1 p3=0
EXEC  #5:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=1,dep=0,og=4,tim=6190451
WAIT #5: nam='SQL*Net message from client' ela= 273 pl=1111838976 p2=1 p3=0
```

А вот этот оператор — неожиданность. Мы сами его не выполняли, и это не рекурсивный SQL-оператор **(dep=O).** Он поступил от клиентского приложения. Все это показывает детали работы утилиты SQL*Plus и пакета **DBMS_OUTPUT. B** файле начального запуска **login.sql** я задал команду **set serveroutput on,** чтобы при запуске утилиты SQL*Plus выдача результатов с помощью пакета **DBMS_OUTPUT** была включена. После каждого выполненного оператора, который мог сгенерировать данные для пакета **DBMS_OUTPUT,** утилита SQL*Plus должна вызвать процедуру **GET_LINES** для получения данных и их выдачи на экран (подробнее о пакете **DBMS_OUTPUT** см. в Приложении А). Мы видим, как утилита SQL*Plus делает этот вызов. Более того, мы видим, что первый параметр, :LINES, фактически является массивом из 25 элементов **(mal=25).** Теперь понятно, что утилита SQL*Plus извлекает из буфера **DBMS_OUTPUT** по 25 строк за раз и выдает их на экран. Тот факт, что можно трассировать действия утилиты SQL*Plus, показателен: значит, мы можем трассировать действия любого программного обеспечения, работающего с СУБД Oracle, и понять, что оно делает.

Наконец, вот последние записи трассировочного файла:

```
XCTEND rlbk=0, rd_only=0
WAIT #0: nam='log file sync' ela= 0 p1=968 p2=0 p3=0
STAT #4 id=1 cnt=1 pid=0 pos=0 obj=0 op='UPDATE EMP  '
STAT #4 id=2 cnt=2 pid=1 pos=1 obj=24767 op='TABLE ACCESS FULL EMP '
```

Запись **XCTEND** (граница транзакции) связана с фиксацией изменений, но мы явно ничего не фиксировали. Утилита SQL*Plus без предупреждений зафиксировала изменения при выходе. В записи **XCTEND** имеются следующие значения:

Поле	*Значение*
rlbk	Флаг отката. Если указано значение 0, значит, транзакция зафиксирована. Значение 1 обозначает откат транзакции.
rd_Only	Флаг только для чтения. Если указано значение 1, транзакция выполнялась в режиме только для чтения. Значение 0 показывает, что изменения были и они зафиксированы (или отменены).

Сразу после записи XCTEND зафиксированы еще какие-то ожидания событий, в данном случае — синхронизации журнального файла. Если обратиться к руководству *Oracle Server Reference* и найти это событие, можно узнать, что значение 988 для параметра **pi** означает, что необходимо записать буфер 988 журнала повторного выполнения, и именно сброса этого буфера на диск мы ждали. Ждать пришлось менее сотой доли секунды, о чем свидетельствует значение **ela=0.**

Последние записи в трассировочном файле — записи **STAT.** Это фактический план выполнения SQL-оператора. Этому плану можно доверять. Более того, для каждого шага плана указано точное количество обработанных строк. Эти записи создаются после закрытия соответствующего курсора. В общем случае это означает, что клиентское приложение должно **завершить работу,** чтобы эти записи появились в файле — выполнения оператора **ALTER SESSION** SET **SQL_TRACE=FALSE** может оказаться недостаточно. Поля этой записи имеют следующие значения:

Поле	*Значение*
cursor #	Номер курсора.
id	Номер строки плана, от 1 до общего количества строк плана.
cnt	Количество строк, прошедших через эту стадию плана.
pid	Идентификатор родительской стадии для данной стадии плана. Используется для корректного отражения иерархии плана с помощью отступов.
pos	Позиция в плане.
obj	Идентификатор соответствующего объекта, при необходимости.
op	Текстовое описание выполняемой операции.

В трассировочных файлах можно обнаружить еще два типа записей. Они описывают ошибки, выявленные при выполнении запроса. Выделяют:

- ошибки разбора (PARSE) — выполнялся недопустимый SQL-оператор;
- ошибки времени выполнения, например дублирование значения ключа индекса, нехватка места и т.д.

Решая различные проблемы, я постоянно обращаюсь к трассировочным файлам, в которых записываются ошибки. Если при использовании готового приложения, средства сторонних производителей и даже команд Oracle выдается невразумительное сообщение об ошибке, имеет смысл выполнить команду с включенной трассировкой SQL_TRACE и посмотреть в трассировочном файле, что на самом деле происходит. Во многих случаях причина проблемы может быть установлена по трассировочному файлу, поскольку все выполненные от имени сеанса SQL-операторы в нем записаны.

Чтобы продемонстрировать эти записи, я выполнил следующий SQL-оператор:

```
tkyte@TKYTE816> create table t (x int primary key);

Table created.

tkyte@TKYTE816> alter session set sql_trace=true;

Session altered.

tkyte@TKYTE816> select * from;
select * from

ERROR at line 1:
ORA-00903: invalid table name


tkyte@TKYTE816> insert into t values (1);

1 row created.

tkyte@TKYTE816> insert into t values (1);
insert into t values (1)
*
ERROR at line 1:
```

```
ORA-00001: unique constraint (TKYTE.SYS_C002207) violated
tkyte@TKYTE816> exit
```

В трассировочном файле я обнаружил:

```
PARSE ERROR #3:len=15 dep=0 uid=69 oct=3 lid=69 tim=7160573 err=903
select * from

PARSING IN CURSOR #3 len=27 dep=0 uid=69 oct=2 lid=69 tim=7161010
hv=1601248092
  ad='32306c0'
insert into t values ( 1 )
END OF STMT
PARSE#3:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=7161010


EXEC#3:c=1,e=9,p=0,cr=9,cu=7,mis=0,r=0,dep=0,og=4,tim=7161019
ERROR #3:err=1 tim=7161019
```

Как видите, поиск проблемного SQL-оператора, ставшего причиной ошибки, с помощью этого метода — тривиальная задача. При этом сразу видно, что не так. Это крайне полезно при поиске ошибки, возникающей, например, глубоко в коде большой хранимой процедуры. Неоднократно я сталкивался с ситуацией, когда ошибка возникала в глубоко вложенном вызове хранимой процедуры, а в приложении использовался обработчик исключительных ситуаций **WHEN OTHERS,** который все исключительные ситуации перехватывал и **игнорировал.** По моему убеждению, обработчик исключительных ситуаций **WHEN OTHERS** использовать вообще нельзя, а все приложения, в которых он используется и при этом не возбуждает исключительную ситуацию повторно, необходимо немедленно удалять — это бомбы замедленного действия. Рано или поздно ошибка произойдет, но она будет перехвачена и проигнорирована, и никто об этом не узнает. Будет казаться, что процедура работает, но на самом деле это не так. В этом случае установка **SQL_TRACE** покажет, что действительно делает процедура, и ошибка будет обнаружена. Останется только разобраться, почему эта ошибка игнорируется. Я также использую трассировку, когда при выполнении операторов выдаются неинформативные сообщения об ошибках. Например, если при обновлении моментального снимка (материализованного представления) выдается сообщение об ошибке **ORA-00942: table or view does not exist,** установка параметра **SQL_TRACE** поможет выяснить истинную причину. Вы можете и не знать всех SQL-операторов, выполняемых от вашего имени при обновлении материализованного представления, и то, какие таблицы они "затрагивают". С помощью **SQL_TRACE** можно легко установить, **какая именно** таблица или представление не существует, и изменить соответствующие права доступа.

Запись **PARSE ERROR** имеет следующий формат:

Поле	*Значение*
len	Длина SQL-оператора
dep	Рекурсивная глубина SQL-оператора

Поле	*Значение*
uid	Схема, от имени которой выполнялся разбор (может отличаться от схемы, для которой определялись права доступа)
oct	Тип команды Oracle
Jid	Схема, с привилегиями которой фактически выполняется оператор
tim	Таймер

err	Код ошибки. Если выполнить следующую команду:
	`tkyte@TKYTE816> EXEC DBMS_OUTPUT.PUT_LINE(SQLERRM(-903));`
	`ORA-00903: invalid table name`
	то можно получить текст сообщения об ошибке.

Формат записи ERROR еще проще:

Поле	*Значение*
cursor #	Номер курсора
err	Код ошибки
tim	Таймер

Итак, в этом разделе достаточно глубоко рассмотрен набор ценнейших средств, работающих во всех средах и всегда доступных. Трассировка при установке **SQL_TRACE** и результирующие отчеты, выдаваемые утилитой **TKPROF** — мощнейшие средства настройки и отладки. Я успешно использовал их для отладки и настройки огромного количества приложений. Их повсеместная доступность (нет ни одной СУБД Oracle, в которой эти средства отсутствуют) в сочетании с возможностями, делает их незаменимыми при настройке.

Зная, как трассировать приложение и интерпретировать результаты трассировки, можно считать, что настройка приложения наполовину завершена. Вторая половина — разобраться, например, почему запрос требует доступа к миллиону блоков или почему он пять минут ждет в очереди. Установка **SQL_TRACE** позволит понять, что именно надо исправлять, а поиск причины проблем обычно сложнее, чем их устранение, особенно если детали реализации приложения неизвестны.

Пакет DBMS PROFILER

Те, кто интенсивно использует язык PL/SQL, с энтузиазмом отнесутся к добавлению в СУБД профилировщика исходного кода. В Oracle 8i полнофункциональный профилировщик исходного кода на языке PL/SQL интегрирован в базу данных. Этот пакет, именуемый **DBMS_PROFILER,** позволяет определить, на какие подпрограммы и пакеты приходится наибольшая доля в общем времени выполнения. Более того, он позволяет анализировать охват кода, т.е. понять, сколько процентов кода приложения проверено в тесте.

В Приложении А я детально описываю использование пакета **DBMS_PROFILER** и интерпретацию получающихся в результате отчетов и данных. Пакет очень легко использовать, и он отлично интегрирован в среду PL/SQL. По ходу работы хранимой процедуры на PL/SQL можно включать и отключать профилирование, сужая фрагмент настраиваемого кода.

При поэтапной настройке приложения я сначала с помощью SQL_TRACE и утилиты **TKPROF** выявляю и исправляю плохо настроенные SQL-операторы. Если все SQL-операторы работают максимально быстро, но необходима дальнейшая настройка, я перехожу к профилированию исходного кода. За исключением действительно неудачных алгоритмов, основное повышение производительности достигается за счет настройки SQL-операторов. Настройка исходного кода PL/SQL дает обычно средние результаты: достаточные, чтобы этим стоило заниматься, но намного меньшие, чем исправление неудачных запросов. Конечно, если использован крайне неэффективный алгоритм, все обстоит иначе.

Еще одно назначение профилировщика — создание отчета об охвате кода в тестовом примере. Это необходимо на этапе тестирования приложения. С помощью этого средства можно составить набор тестов, покрывающих весь код в ходе тестирования. Хотя это и не гарантирует отсутствие ошибок в коде, но позволяет избежать очевидных упущений.

Средства контроля и отладки

Обеспечение средств для контроля и отладки — жизненно важная задача, особенно в большом приложении с множеством компонентов. Чем сложнее приложение и чем больше компонентов оно включает, тем сложнее найти компонент, снижающий производительность. *Средства контроля и отладки* (instrumentation) — это части кода, обеспечивающие возможность журнализации, которую можно избирательно включать для определения того: а) что делает программа и б) как долго она это делает. Все, от простейшего процесса до самого хитроумного алгоритма, должно быть снабжено полноценными средствами контроля и отладки. Да, эти средства требуют дополнительных расходов ресурсов, даже если сообщения в журнал не записываются, но невозможность определить, где возникает проблема, намного хуже, чем небольшое падение производительности при наличии этих средств.

В начале главы я рассказывал о недавно выполненной мною настройке. Архитектура приложения была, мягко говоря, сложной. Пользовательский интерфейс в Web-браузере взаимодействовал с группой Web-серверов через весьма строгий брандмауэр. Web-серверы поддерживали страницы Java Server Pages (JSP). Эти страницы через пул подключений на сервере приложений взаимодействовали с базой данных, где выполнялся код SQL и PL/SQL. Кроме того, они с помощью CORBA обращались к еще одному серверу приложений для взаимодействия с существующей системой. Они были лишь составной частью большего приложения. В системе были еще пакетные задания, фоновые процессы, обработка очередей и другие готовые компоненты. Мне пришлось начинать, абсолютно ничего не зная о приложении, но что еще хуже, мало кто там вообще знал

хоть что-нибудь о системе в целом: все знали только свои задачи и компоненты. Представить себе ситуацию в целом всегда сложно.

Поскольку ни один из компонентов не был снабжен средствами контроля и отладки, для выявления проблемы пришлось использовать представленные ранее средства СУБД. Трассировка с помощью SQL_TRACE не показала ничего подозрительного (фактически, она показала, что SQL-операторы выполняются отлично). Профилировщик тоже не дал ничего, кроме подтверждения, что PL/SQL-код тоже работает отлично. Представления динамической производительности V$ в базе данных подтвердили, что все в порядке. С помощью этих средств мы просто доказали, что замедление не связано с работой СУБД. Но выявить проблему они не помогли. Она была вне базы данных, где-то между браузером и базой данных. К сожалению, этот промежуток представлял собой жуткую смесь страниц JSP, компонентов EJB, пулов подключений и CORBA-вызовов. Нам оставалось только снабдить код средствами контроля и отладки, чтобы выявить медленно работающий компонент. Нельзя было просто запустить программу под отладчиком, как "в старые добрые времена". Приходилось работать с десятками компонентов и фрагментов кода, связанными по сети — именно эти компоненты и должны были нам указать, какой именно из них работает медленно.

В конце концов мы этот компонент обнаружили. Это был CORBA-вызов существующей системы, который выполнялся для генерации практически каждой страницы. Только создав журнальные файлы с временными метками, мы смогли это обнаружить. Оказалось, что это не проблема базы данных и даже не проблема приложения, но пользователям от этого легче не стало; для них не важно, чей код работает медленно. Жаль было потраченных на обнаружение причин проблемы недель.

Способ включения средств контроля и отладки зависит от используемого языка программирования. Например, в языке PL/SQL я использую специально разработанный для этого пакет DEBUG. Он реализует стандартный механизм журнализации для создания журнальных файлов в любой подпрограмме PL/SQL. Достаточно включить вызовы debug.f в код приложения следующим образом:

```
create function foo ...
as

begin
      debug.f('Enter procedure f00');
      if (some_condition) then
           l_predicate := 'x=l';
      end if;

      debug.f("Going to return the predicate "%s"', l_predicate);
      return l_predicate;
end;
```

и автоматически в журнал вносятся записи вида:

```
011101 145055 ( FOO,  6) Enter procedure foo
011101 145056 ( FOO, 11) Going to return the predicate "x=l"
```

При этом в журнал автоматически добавляется дата (01/11/2001) и время (14:50:55), а также информация о том, в какой процедуре/пакете и в какой строке был вызов. Можно

включать и отключать трассировку в любом модуле или наборе модулей. Это не только полезное средство отладки — по трассировочным файлам легко понять, что именно работало долго. Утилиту **DEBUG** мы еще рассмотрим в главе 21, где она многократно упоминается.

Если бы все приложения и их компоненты имели подобные средства, искать причины снижения производительности стало бы намного проще. В системе без таких средств искать их — все равно, что иголку в стоге сена. Даже хуже, поскольку при поисках иголки вряд ли вокруг будут крутиться советчики, пытающиеся высказать свое мнение о причине проблем и направить поиски по этому пути.

Еще раз повторю: все компоненты приложения, даже не связанные с базой данных, должны быть снабжены средствами контроля и отладки, особенно в современных средах, где редко встретишь приложение, взаимодействующее с единственным сервером. С учетом роста популярности Web-приложений, в особенности, использующих сложные распределенные архитектуры, определение источников проблем намного сложнее их устранения. Снабдив с самого начала код средствами контроля и отладки, мы реализуем защитный подход к программированию, который положительно скажется в дальнейшем. Я гарантирую, что вы никогда не пожалеете о включении в приложение средств контроля и отладки, жалеть придется, если вы этого не сделаете.

Я настоятельно рекомендую оставить средства отладки в коде готового приложения. **Не удаляйте средства отладки в реально работающем приложении.** Именно там они наиболее полезны! Я обычно подставляю пустую реализацию тела пакета **DEBUG** (где все функции сразу делают возврат). При этом если необходим трассировочный файл, я подставляю реальное тело пакета **DEBUG,** трассирую все, что нужно, а затем подставляю снова пустую реализацию. Удаление кода из производственного приложения "из соображений производительности" существенно снижает его полезность. Посмотрите на саму СУБД: путем установки огромного количества *событий* служба технической поддержки Oracle может получить огромный объем диагностических данных из производственной базы данных клиента. Разработчики ядра СУБД поняли, что отсутствие трассировочного кода в производственной системе значительно хуже, чем возможные дополнительные расходы ресурсов.

Набор утилит StatsPack

Мы уже рассмотрели средства настройки приложений. Трассировка с помощью **SQL_TRACE,** пакет **DBMS_PROFILER,** добавление средств контроля и отладки — все это обеспечивает настройку на уровне приложения. Если есть уверенность, что приложение работает максимально хорошо, имеет смысл разобраться с работой всего экземпляра базы данных, оценить его производительность в целом. Именно здесь перекрываются служебные обязанности и размывается грань между функциями администратора базы данных и разработчика приложений. Администратор базы данных должен найти причину замедления, но устранять ее зачастую приходится разработчику. Совместная работа в данном случае обязательна.

Раньше для оценки работы экземпляра использовались средства **UTLBSTAT** и **UTLESTAT** (начать сбор статистики и закончить сбор статистики). Сценарий **UTLBSTAT**

делал моментальный снимок многих представлений динамической производительности V$. Затем с помощью сценария UTLESTAT создавался отчет по "начальным" и "конечным" значениям в таблицах V$. Полученная статистическая информация обрабатывалась и выдавалась для изучения в виде простого текстового отчета. Начиная с Oracle8i, сценарии **BSTAT/ESTAT** формально были заменены набором утилит StatsPack. Этот набор инструментальных средств намного превосходит прежние возможности **BSTAT/ESTAT.** Наиболее существенным добавлением стала возможность сохранять значения представлений V$ в хронологическом порядке. Т.е. вместо удаления статистической информации после формирования отчета утилиты StatsPack позволяют сохранить данные и генерировать отчеты позже, при необходимости. При использовании средств **BSTAT/ESTAT,** например, невозможно было генерировать ежедневные отчеты и почасовой отчет для каждого дня недели. С помощью средств StatsPack можно установить почасовой сбор статистической информации (что мало влияет на работу системы) и генерировать отчеты, сравнивающие два "моментальных снимка". Таким образом, можно создать отчет и за любой час, и за любой день недели.

Кроме гибкости при создании отчетов, средства StatsPack обеспечивают более полный набор регистрируемых данных. В этом разделе я собираюсь описать установку утилит StatsPack, сбор данных и, самое главное, анализ получаемых отчетов.

Установка утилит StatsPack

Пакет утилит StatsPack должен устанавливаться при подключении как **INTERNAL** или, еще лучше, как **SYSDBA (CONNECT SYS/CHANGE_ON_INSTALL AS SYSDBA),** хотя в сценариях все равно будет выполняться **CONNECT INTERNAL.** Для успешной установки необходимо выполнить эту операцию. Очевидно, придется попросить сделать это администратора базы данных или администраторов сервера.

Если можно подключиться как **INTERNAL,** установка StatsPack — тривиальна. Необходимо просто выполнить сценарий **statscre.sql** в версии 8.1.6 или **spcreate.sql** в версии 8.1.7. Они находятся в каталоге **[ORACLE_HOME]\rdbms\admin** после подключения как **INTERNAL** с помощью SQL*Plus. Процесс установки выглядит примерно так:

```
C:\oracle\RDBMS\ADMIN>sqlplus internal

SQL*PLUS: Release 8.1.6.0.0 - Production on Sun Mar 18 11:52:32 2001

(c) Copyright 1999 Oracle Corporation.  All rights reserved.


Connected to:
Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option
JServer Release 8.1.6.0.0 - Production

sys@TKYTE816> @statscre
. . . Installing Required Packages
```

Перед запуском сценария **statscre.sql** надо знать следующее.

- Имя стандартного табличного пространства для пользователя **PERFSTAT,** который будет создан.
- Имя временного табличного пространства для этого пользователя.
- В каком табличном пространстве создавать объекты StatsPack. Этот параметр не запрашивается в версии 8.1.7 — только в 8.1.6. В данном табличном пространстве должно быть свободное место для примерно 60 экстентов (так что реальный размер будет зависеть от стандартного размера экстента).

Сценарий будет запрашивать эту информацию в ходе выполнения. Если при вводе сделана ошибка или установка была прервана по ходу, надо с помощью сценариев **spdrop.sql** (8.1.7 и далее) или **statsdrp.sql** (8.1.6 и ранее) удалить пользователя и все уже установленные представления, прежде чем снова устанавливать StatsPack. При установке StatsPack будет создано три файла с расширением **.lis** (их имена выдаются на экран в ходе установки). Просмотрите эти файлы и убедитесь, что при установке не было ошибок. Все должно устанавливаться без проблем, если были указаны подходящие имена табличных пространств (и нет пользователя **PERFSTAT).**

Теперь, после установки StatsPack, осталось собрать хотя бы два набора данных. Самый простой способ сделать это — с помощью пакета **STATSPACK,** принадлежащего пользователю **PERFSTAT:**

```
perfstat@DEV2.THINK.COM> exec statspack.snap

PL/SQL procedure successfully completed.
```

Теперь надо подождать некоторое время, дав системе поработать, и сделать еще один моментальный снимок. При наличии данных за два момента времени, создать отчет тоже просто. Для этого надо выполнить сценарий **statsrep.sql (8.1.6)** или **spreport.sql** (8.1.7). Это сценарий для утилиты SQL*Plus, который надо выполнять от имени пользователя **PERFSTAT** (по умолчанию его пароль — **PERFSTAT,** но его надо изменить сразу же после установки!). Формат отчета несколько изменился при переходе с версии 8.1.6 на 8.1.7, причем формат версии 8.1.7 мне нравится больше; его мы и рассмотрим. Для создания отчета достаточно выполнить команду:

```
perfstat@ORA8I.WORLD> @spreport


   DB Id    DB Name      Inst Num  Instance
------------ ------------ --------- ------------
 4080044148  ORA8I               1  ora8i


Completed Snapshots
                    Snap                          Snap
Instance  DB Name      Id  Snap Started          Level  Comment

ora8i     ORA8I         1  18 Mar 2001 12:44        10
                        2  18 Mar 2001 12:47        10


Specify the Begin and End Snapshot Ids

Enter value for begin_snap:
```

Выдается список моментов времени, для которых собрана информация, и предлагается выбрать два из них для сравнения. Затем будет сгенерировано стандартное имя отчета и предложено принять его или задать новое. После этого генерируется отчет. Ниже представлен отчет StatsPack версии 8.1.7, по разделам, с комментариями о том, на что обращать внимание и как интерпретировать результаты.

```
STATSPACK report for
DB Name         DB Id    Instance  Inst Num Release    OPS  Host
------------ ----------- ------------ -------- ----------- --- ------------
ORA8I        4080044148  ora8i            1  8.1.6.2.0 NO   aria
                         Snap Id      Snap Time      Sessions
                         ------- ------------------ --------
    Begin Snap:                1 18-Mar-01 12:44:41       22
      End Snap:                3 18-Mar-01 12:57:23       22
       Elapsed:                    12.70 (mins)

Cache Sizes
~~~~~~~~~~~
          db_block_buffers:    16384       log_buffer:     512000
          db_block_size:        8192  shared_pool_size:  102400000
```

Первая часть отчета носит справочный характер. Здесь показано, по какой базе данных был создан отчет, в частности имя и идентификатор базы данных. В вашей среде они должны быть уникальными. В поле **Instance** указан идентификатор **SID** для базы данных. Эти три параметра помогут разобраться, по какой именно базе данных был сгенерирован отчет. Одной из проблем в старых отчетах BSTAT/ESTAT было то, что они не включали никакой идентификационной информации. Неоднократно я получал отчет для анализа, а потом оказывалось, что он делался не на том сервере, где возникла проблема. Теперь такое не случится. Далее идет информация о моментах времени сбора данных и интервале между этими моментами. Для многих странно, что эти интервалы не обязательно должны быть большими — представленный выше отчет покрывает период продолжительностью 13 минут. Важно лишь, чтобы в этот период шла обычная работа с базой данных. Отчет StatsPack для периода продолжительностью 15 минут не менее показателен, чем для периода в один или несколько часов. Чем больше охватываемый период, тем сложнее прийти к определенным выводам по полученным числовым данным. Завершается этот раздел общей информацией о конфигурации сервера. Можно узнать параметры основных компонентов области SGA:

```
Load Profile
~~~~~~~~~~~~                          Per secona       rar Transaction
                                 ---------------       ---------------
                  Redo size:           5,982.09             13,446.47
              Logical reads:           1,664.37              3,741.15
              Block changes:              17.83                 40.09
             Physical reads:              15.25                 34.29
            Physical writes:               5.66                 12.73
                 User calls:               3.28                  7.37
                     Parses:              16.44                 36.96
                Hard parses:               0.17                  0.37
```

Sorts:	**2.95**	**6.64**
Logons:	**0.14**	**0.32**
Executes:	**30.23**	**67.95**
Transactions:		**0.44**

В этом разделе компактно представлен большой объем информации. Можно увидеть, какой объем данных повторного выполнения **(REDO)** генерировался в среднем за секунду и за транзакцию (в данном случае генерировалось от 5 до 6 Кбайт информации повторного выполнения в секунду). Средняя транзакция генерировала около 13 Кбайт данных повторного выполнения. Следующий блок информации связан с логическим и физическим вводом/выводом. Показано, что около 1 процента логических считываний потребовали чтения физического — это очень хорошо. Также показано, что в среднем транзакции выполняли почти 4000 логических считываний. Много это или мало, зависит от системы. В моем случае работало несколько больших фоновых заданий, так что такой объем чтения вполне приемлем.

Далее идет действительно важная информация — статистика по разборам. Она свидетельствует, что в среднем выполнялось по 16 разборов в секунду, причем выполнялось в среднем по 0,17 жестких разбора в секунду (разбирались никогда ранее не встречавшиеся операторы SQL). Примерно раз в шесть секунд система разбирала абсолютно новый оператор SQL. Это неплохо. Однако если бы в этом столбце за период, равный нескольким дням, сохранилось значение ноль, что свидетельствует о хорошей настройке системы, я был бы доволен. С некоторого момента все операторы SQL должны находится в разделяемом пуле.

```
  % Blocks changed per Read:     1.07    Recursive Call %:    97.39
Rollback per transaction %:     0.29       Rows per Sort:   151.30
```

В следующем разделе приведен ряд интересных показателей. Параметр % **Blocks Changed per Read** показывает, что 99 логических чтений пришлись на блоки, которые только **считывались,** но не изменялись. Система изменяла только 1 процент прочитанных блоков. Процент рекурсивных вызовов **(Recursive Call** %) очень высок — более 97. Это не означает, что 97 процентов SQL-операторов в моей системе выполняются в ходе "управления пространством" или разбора. Если вернуться к анализу исходного трассировочного файла в разделе, посвященном **SQL_TRACE,** там было показано, что операторы SQL, выполняющиеся в PL/SQL, также считаются "рекурсивными". В моей системе почти все, кроме модуля **mod_plsql** (это модуль Web-сервера Apache) и редких пакетных заданий, выполняется с помощью PL/SQL; все остальное написано на языке PL/SQL. Поэтому я бы удивился низкому значению **Recursive Call** %.

Процент отмененных транзакций **(Rollback per transaction** %) — очень низкий, и это хорошо. Откат — весьма дорогостоящая операция. Сначала мы что-то делаем, и на это тратятся ресурсы. Затем мы отменяем сделанное, и на это тоже требуются ресурсы. Много ресурсов потрачено без всякого результата. Если большинство транзакций откатывается, значит, основные ресурсы сервера уходят на то, чтобы что-то делать и тут же отменять сделанное. Надо разобраться, **чем** вызван такой объем отката и как можно переделать приложение, чтобы это изменить. В рассматриваемой системе лишь одна из примерно 345 транзакций откатывается — это приемлемо.

```
Instance Efficiency Percentages  (Target 100%)

            Buffer Nowait %:   100.00        Redo NoWait %:   100.00
           Buffer  Hit   %:    99.08     In-memory Sort %:    99.60
           Library Hit   %:    99.46         Soft Parse %:    98.99
         Execute to Parse %:    45.61          Latch Hit %:   100.00
Parse CPU to Parse Elapsd %:    87.88      % Non-Parse CPU:   100.00
```

Далее показана эффективность работы экземпляра, **Instance Efficiency Percentages.** Утверждается, что по этим показателям надо стремиться к 100 процентам, и это почти правда. Единственным исключением, по моему мнению, является показатель Execute **to Parse.** Он показывает, сколько раз в среднем выполнялся разобранный оператор. В системе, где оператор разбирается, выполняется один раз и больше никогда не выполняется в том же сеансе, этот показатель будет равен нулю. В представленном примере каждый разобранный оператор выполнялся в среднем 1,8 раза (отношение почти два к одному). Хорошо это или плохо, зависит от особенностей системы. В моей системе для всех приложений используется модуль **mod_plsql.** Создается сеанс, выполняется хранимая процедура, формирующая Web-страницу, и сеанс завершается. Если только в одной хранимой процедуре один и тот же SQL-оператор не выполняется несколько раз, отношение количества выполнений к количеству разборов будет небольшим. С другой стороны, если используется клиент-серверное приложение или подключение к базе данных выполняется с сохранением состояния (например, через интерфейс сервлетов), этот коэффициент (он вычисляется как отношение выполнений без разбора к выполнениям с разбором — *прим. научн. ред.)* действительно должен быть близок к 100 процентам. Я понимаю, однако, что с учетом используемой архитектуры, существенного повышения этого коэффициента в своей системе я добиться не смогу.

Для меня наиболее важными являются показатели разбора, я сразу обращаю на них внимание. Коэффициент **Soft Parse** показывает процент мягких разборов по отношению к общему количеству выполненных разборов. 99 процентов разборов в моей системе были мягкими (повторно использовались планы выполнения запросов из разделяемого пула). Это хорошо. Если процент мягких разборов — низкий, значит, в системе не используются связываемые переменные. Я считаю, что этот показатель должен быть очень близок к 100 процентам, независимо от использовавшихся для разработки приложений средств и методов. Низкое значение показывает, что напрасно тратятся ресурсы и появляются конфликты доступа. Затем надо обратить внимание на коэффициент **Parse CPU to Parse Elapsd. В** данном случае он имеет значение около 88 процентов. Это маловато; мне надо над этим поработать. В данном случае на каждую секунду процессорного времени, потраченного на разбор, приходится около 1,13 секунды реального времени. Это означает, что часть времени уходит на ожидание ресурсов. Если бы этот коэффициент имел значение 100 процентов, то реальное время было бы равно процессорному, т.е. при обработке никто бы никого не ждал. Наконец, коэффициент **Non-Parse** CPU показывает отношение времени, потраченного на реальную работу, к общему времени, включая разбор операторов. В отчете это значение вычисляется как **round(100*(l-PARSE_CPU/ TOT_CPU),2).** Если общее время работы, **TOT_CPU,** намного превосходит время разбора, **PARSE_CPU** (как и должно быть), этот коэффициент будет очень близок к 100

процентам, как у меня. Это хорошо и показывает, что основное время работы компьютера ушло на **выполнение,** а не на **разбор** запросов.

Подводя итоги по предыдущему разделу, я бы рекомендовал сократить количество жестких разборов. Скорее всего в системе есть еще несколько операторов, не использующих связываемые переменные (каждые шесть секунд появляется новый запрос). Это, в свою очередь, сократит общее количество выполняемых разборов, поскольку при жестком разборе выполняется множество рекурсивных SQL-операторов. Убрав лишь один жесткий разбор, можно сократить также и количество мягких разборов. Все остальные показатели в данном разделе выглядят вполне приемлемыми. Рассмотренная первая часть отчета StatsPack мне нравится больше всего — она дает общее представление об относительном "здоровье" системы. Теперь переходим к остальной части отчета:

```
Shared Pool Statistics      Begin     End

          Memory Usage %:   75.03     75.26
% SQL with executions>1:    79.18     78.72
% Memory for SQL w/exec>1: 74.15     73.33
```

Этот маленький фрагмент дает некоторую информацию об использовании разделяемого пула. Рассмотрим показатели более детально:

- **Memory** Usage. Процент использования разделяемого пула. Со временем это значение должно стабилизироваться в диапазоне от 70 с небольшим до 90 процентов. Если процент использования слишком низкий, значит, память тратится напрасно. Если процент слишком высок, значит, происходит вытеснение компонентов разделяемого пула как устаревших, что приводит к жесткому разбору SQL-операторов при повторном выполнении. При правильно подобранном размере разделяемого пула должно использоваться от 75 до не более чем 90 процентов его пространства.

- **SQL with executions>1.** Этот показатель определяет, сколько SQL-операторов в разделяемом пуле выполнялось больше одного раза. К оценке его значения в системах, работающих циклически, с разными задачами в течение дня (например, задачи класса оперативной обработки транзакций в рабочее время и задачи систем поддержки принятия решений по ночам) надо подходить вдумчиво. За изучаемый период многие SQL-операторы в разделяемом пуле могут не выполниться повторно только потому, что в этот период не выполнялись посылающие их процессы. Только если в системе постоянно выполняются одни и те же SQL-операторы, этот показатель может приблизиться к 100 процентам. В данном случае около 80 процентов SQL-операторов в разделяемом пуле использовалось более одного раза за рассмотренный период продолжительностью 13 минут. Остальные 20 процентов моей системе, вероятно, просто не понадобилось повторно выполнять.

- **Memory for SQL w/exec>l.** Этот показатель определяет, сколько памяти используют часто выполняемые операторы SQL по сравнению с теми, которые выполняются редко. В общем случае его значение будет очень близким к проценту нео-

днократно выполнявшихся операторов SQL, если только нет запросов, требующих слишком много памяти. Этот показатель мне не кажется особенно полезным.

Итак, в общем случае в стабильном состоянии должно использоваться от 75 до 85 процентов разделяемого пула. SQL-операторы, выполнявшиеся более одного раза, должны составлять около 100 процентов, если в отчете StatsPack рассматривается достаточно продолжительный период, охватывающий все циклы работы. На этот показатель как раз влияет продолжительность периода между наблюдениями. Чем больший период рассматривается, тем больше должно быть это значение.

Теперь переходим к следующему разделу:

```
Top 5 Wait Events
                                                     Wait     % Total
Event                               Waits    Time  (cs)    Wt  Time
------------------------------  -------  ----------  -----------
SQL*Net more data from dblink     1,661         836        35.86
control file parallel   write       245         644        27.63
log file sync                       150         280        12.01
db file scattered read            1,020         275        11.80
db file sequential read             483         165         7.08

Wait Events for DB: ORA8I  Instance: ora8i  Snaps: 1 -3
-> cs - centisecond -  100th of a second
-> ms - millisecond - 1000th of a second
-> ordered by wait time desc, waits desc (idle events last)
```

Вот ваша ближайшая цель: события, замедляющие работу намного больше всего остального. Сначала надо посмотреть на значение **Wait Time,** чтобы понять, стоит ли время ожидания того, чтобы настраивать систему для его уменьшения. Например, за 13 минут я потратил 8,36 секунд в ожидании данных из удаленной базы **(data from a dblink).** Стоил ли тратить время на поиски и "устранение" причины? В данном случае, я считаю, не стоит: среднее ожидание составило 0,004 секунды. Более того, я знаю, что работает фоновый процесс, выполняющий сложную операцию с удаленной базой данных, так что с учетом всех факторов время ожидания получилось весьма небольшим.

Итак, пусть найдено событие, требующее внимания. Сначала нужно понять, что это за событие. Например, если посмотреть описание события **log file sync** в руководстве *Oracle Reference Manual,* то можно узнать, что:

> *Когда пользовательский сеанс фиксирует транзакцию, информация повторного выполнения должна быть сброшена в файл журнала повторного выполнения. Пользовательский сеанс выдает задание процессу LGWR на запись буфера журнала повторного выполнения в файл журнала. Когда процесс LGWR завершит запись, он уведомляет об этом пользовательский сеанс.*
>
> ***Wait Time:*** *время ожидания включает время записи буфера журнала и время уведомления.*

Теперь, когда понятно, чего именно пришлось ждать, можно придумать, как от этого ожидания избавиться. Когда ожидается синхронизация файла журнала, надо настра-

ивать работу процесса LGWR. Чтобы уменьшить время ожидания можно использовать более быстрые диски, генерировать меньше информации повторного выполнения, снизить конфликты доступа к дискам, содержащим журналы, и т.д. Найти причину ожидания — одно дело, устранить ее — совсем другое. В Oracle измеряется время ожидания более 200 событий, причем ни для одного из них нет простого способа сократить время ожидания.

Не стоит забывать, что ждать чего-нибудь придется всегда. Если устранить одно препятствие, появится другое. Нельзя вообще избавиться от длительного ожидания событий — всегда придется чего-то ждать. Настройка "для максимально быстрой работы" может продолжаться бесконечно. Всегда можно сделать так, чтобы скорость работы возросла на один процент, но время, которое необходимо затратить на обеспечение каждого последующего процента прироста производительности, растет экспоненциально. Настройкой надо заниматься при наличии конкретной конечной цели. Если нельзя сказать, что настройка закончена, если достигнут показатель X, где X можно измерить, значит, вы напрасно тратите время.

Следующий раздел отчета:

```
Wait Events for DB: ORA8I  Instance: ora8i  Snaps: 1 -3
-> cs - centisecond -  100th of a second
-> ms - millisecond - 1000th of a second
-> ordered by wait tine desc, waits desc  (idle events last)
```

Event	Waits	Timeouts	Total Time	Avg Wait (cs)	wait (ms)	Waits /txn
SQL*Net more data from dblin	1,861	0	836	4	5.5
control file parallel write	245	0	644	26	0.7
log file sync	150	0	280	19	0.4
db file scattered read	1,020	0	275	3	3.0
db file sequential read	483	0	165	3	1.4
control file sequential read	206	0	44	2	0.6
SQL*Net message from dblink	51	0	35	7	0.2
refresh controlfile command	21	0	28	13	0.1
log file parallel write	374	0	14	0	1.1
latch free	13	10	3	2	0.0
SQL*Net more data to client	586	0	2	0	1.7
single-task message	1	0	2	20	0.0
direct path read	716	0	1	0	2.1
direct path write	28	0	1	0	0.1
file open	28	0	1	0	0.1
SQL*Net message to dblink	51	0	0	0	0.2
db file parallel write	24	0	0	0	0.1
LGWR wait for redo copy	3	0	0	0	0.0
file identify	1	0	0	0	0.0
SQL*Net message from client	2,470	0	1,021,740	4137	7.3
virtual circuit status	25	25	76,825	30730	0.1
pipe get	739	739	76,106	1030	2.2
SQL*Net more data from clien	259	0	3	0	0.8
SQL*Net message to client	2,473	0	0	0	7.3

показывает все ожидания событий клиентами, произошедшие за рассматриваемый период. Кроме информации, представленной в разделе Top 5, показано среднее время ожидания в тысячных долях секунды, а также, сколько раз транзакция ожидала этого события. Это помогает найти существенные события. Обращаю ваше внимание, что в этом разделе множество событий надо игнорировать. Например, ожидание события SQL*Net message from client можно игнорировать в тех системах, где клиенту необходимо время на размышление. Это время, в течение которого клиент не обращался к базе данных с запросами (с другой стороны, если подобные ожидания происходят при загрузке данных, значит, клиент недостаточно быстро передает данные, и это уже проблема). В нашем случае, однако, это ожидание означает, что клиент был подключен, но не делал никаких запросов. В заголовке раздела сказано, что записи ожиданий событий, связанные с простоями, приведены в конце. Все события, начиная с SQL*Net message from client и ниже, связаны с простоями: процесс ждал, пока к нему обратятся. В большинстве случаев все эти ожидания можно игнорировать.

```
Background Wait Events for DB: ORA8I  Instance: ora8i  Snaps: 1  -3
-> ordered by trait tine desc, waits desc (idle events last)
```

Event	Waits	Total Timeouts	Time	Wait (cs)	Avg wait (ms)	Waits /txn
control file parallel write	245	0		644	26	0.7
control file sequential read	42	0		25	6	0.1
log file parallel write	374	0		14	0	1.1
db file parallel write	24	0		0	0	0.1
LGHR wait for redo copy	3	0		0	0	0.0
rdbms ipc message	1,379	741		564,886	4096	4.1
smon timer	3	3		92,163		0.0
pmon timer	248	248		76,076	3068	0.7

Представленный выше раздел отчета StatsPack показывает ожидания событий "фоновыми" процессами (DBWR, LGWR и т.д.). И в этом разделе записи для событий, связанных с простоями, приведены в конце, и тоже, в общем случае, могут быть проигнорированы. Раздел пригодится при настройке экземпляра в целом, чтобы понять, чего именно ждут фоновые процессы. Что именно замедляет работу сеанса, определить несложно — мы уже многократно делали это в примерах, посвященных использованию связываемых переменных, и в других разделах: достаточно выполнить запрос к представлению V$SESSION_EVENT. Этот фрагмент отчета показывает, каких событий ожидают фоновые процессы, во многом аналогично тому, как ожидания представлены для отдельных сеансов.

```
SQL ordered by Gets for DB: ORA8I  Instance: ora8i  Snaps: 1 -3
-> End Buffer Gets Threshold:    10000
-> Note that resources reported for PL/SQL includes the resources used by
    all SQL statements called within the PL/SQL code.  As individual SQL
    statements are also reported, it is possible and valid for the summed
    total % to exceed 100
```

```
  Buffer Gets     Executions   Gets per Exec    % Total   Hash Value
---------------  ---------- --------------- ---------- ----------
         713,388          1       713,388.0       56.2   1907729738

BEGIN sys.sync_usera.do_it; END;

         485,161          1       485,161.0       38.3   1989876028
SELECT DECODE(SA.GRANTEE#,1,'PUBLIC',U1.NAME) "GRANTEE",U2.NAME
"GRANTED_ROLE",DECODE(OPTION$,1,'YES','NO') "ADMIN_OPTION" FRO
M SYSAUTH$@ORACLE8.WORLD SA,DEFROLE$@ORACLE8.WORLD UD,USER$@ORAC
LE8.WORLD U1,USER$@ORACLE8.WORLD U2 WHERE SA.GRANTEE# = UD.USER
# (+) AND SA.PRIVILEGE# = UD.ROLE# (+) AND U1.USER* = SA.G

         239,778          2       119,889.0       18.9    617705294
BEGIN statspack.snap(10); END;
```

В этом разделе представлены "основные" операторы SQL. Все SQL-операторы упорядочены по убыванию показателя **Buffer** Gets, другими словами, по убыванию выполняемых логических операций ввода/вывода. Как сказано в комментарии в начале отчета, количество прочитанных буферов для программной единицы **PL/SQL** равно сумме обращений к буферам **всех** SQL-операторов, выполненных в соответствующем блоке кода. Поэтому часто в начале списка можно обнаружить PL/SQL-процедуры — показатели отдельных составляющих операторов для них суммируются.

В нашем случае первой указана PL/SQL-процедура **sync_users.do_it.** Она затрагивает более 700000 блоков при каждом выполнении. Хорошо это или плохо, по данному фрагменту отчета не понятно. Здесь представлены только факты — никаких оценок. Я знаю, что sync_users — большое фоновое задание, синхронизирующее словари данных двух баз и гарантирующее, что пользователь, созданный в одной базе данных, создается и в другой, а также что совпадают все роли и пароли пользователей. Вполне понятно, что она обрабатывает большой объем информации. Как оказалось, именно это задание и ждало поступления информации из удаленной базы данных (это ожидание мы обнаружили ранее).

```
SQL ordered by Reads for DB: ORA8I  Instance: oxa8i  Snaps: 1 -3
-> End Disk Reads Threshold:     1000

Physical Reads   Executions   Reads per Exec   % Total   Hash Value

         8,484          1          8,484.0      73.0   1907729738
BEGIN sys.sync_users.do_it; END;

         2,810          2          1,405.0      24.2    617705294
BEGIN statspack.snap(10); END;
```

Этот раздел очень похож на предыдущий, но вместо логического ввода/вывода он информирует о физическом вводе/выводе. В нем показаны SQL-операторы, наиболее активно физически **читающие** данные. Именно на эти запросы и процессы надо обратить внимание, если система не справляется с объемом ввода/вывода. Процедуру sync_users, видимо, надо настроить — она является основным потребителем ресурсов дисковой подсистемы.

```
SQL ordered by Executions for DB: ORA8I  Instance: ora8i  Snaps: 1 -3
-> End Executions Threshold:      100

   Executions  Rows Processed  Rows per Exec  Hash Value
-------------- --------------- -------------- ----------
        2,583               0            0.0  4044433098
SELECT TRANSLATE_TO_TEXT    FROM WWV_FLOW_DYNAMIC_TRANSLATIONS$
WHERE TRANSLATE_FROM_TEXT =  :b1  AND TRANSLATE_TO_LANG_CODE =  :b
2

        2,065           2,065            1.0  2573952486
SELECT DISPLAY_NAME    FROM WWC_PEOPLE_LABEL_NAMES   WHERE LABEL_N
AME =  :b1
```

Эта часть отчета об "основных" SQL-операторах показывает, какие операторы выполнялись чаще всего за рассматриваемый период. Эта информация может пригодиться для выявления ряда наиболее часто выполняемых запросов с целью изменения алгоритмов работы приложений так, чтобы эти запросы выполнялись не так часто. Возможно, запрос выполняется в цикле, а мог бы выполняться один раз за пределами цикла — несложное изменение алгоритма может уменьшить количество выполнений запроса. Даже если запрос выполняется мгновенно, его выполнение миллион раз требует существенного времени.

```
SQL ordered by Version Count for DB: ORA8I  Instance: ora8i  Snaps: 1 -3
-> End Version Count Threshold:          20

Version
  Count    Executions   Hash Value

     21           415    451919557
SELECT SHORTCUT_NAME,ID    FROM  WWV_FLOW_SHORTCUTS  WHERE  FLOW_ID
 =  :b1   AND  (:b2  IS NULL   OR SHORTCUT_NAME =  :b2  )  AND NOT EXIST
S   (SELECT 1    FROM WWV_FLOW_PATCHES   WHERE FLOW_ID =  :b1   AND I
D = BUILD_OPTION   AND PATCH_STATUS =  'EXCLUDE'  )   ORDER BY SHORT
CUT_NAME, SHORTCUT_CONSIDERATION_SEQ

     21           110   1510890808
SELECT DECODE(:b1,1,ICON_IMAGE,2,ICON_IMAGE2,3,ICON_IMAGE3) ICON
_IMAGE,DECODE(:b1,1,ICON_SUBTEXT,2,ICON_SUBTEXT2,3,ICON_SUBTEXT3
) ICON_SUBTEXT,ICON_TARGET,ICON_IMAGE_ALT,DECODE(:b1,1,ICON_HEIG
HT,2,NVL(ICON_HEIGHT2,ICON_HEIGHT),3,NVL(ICON_HEIGHT3,ICON_HEIGH
T)) ICON_HEIGHT,DECODE(:b1,1,ICON_WIDTH,2,NVL(ICON_WIDTH2,ICON_H
```

В этом разделе показаны SQL-операторы по убыванию количества экземпляров одного и того же оператора в разделяемом пуле. Причин наличия нескольких экземпляров одного и того же SQL-оператора в разделяемом пуле может быть много. Вот некоторые из них.

- Разные пользователи выполняли один и тот же SQL-оператор, но обращается он к разным таблицам.
- Тот же запрос выполняется в принципиально отличающейся среде, например, с другим режимом оптимизации.

- Для перезаписи запроса используется механизм тщательного контроля доступа (Fine Grained Access Control). Каждая версия в разделяемом пуле на самом деле выполнялась как существенно отличающийся запрос.
- Клиент использует связываемые переменные разных типов или размеров: одна программа связывает запрос с текстовой строкой длиной 10 символов, а другая — со строкой длиной 20 символов. В результате тоже получается новая версия SQL-оператора.

Следующий пример показывает, как в разделяемом пуле получается несколько версий одного и того же SQL-запроса. Мы начнем с очистки разделяемого пула, чтобы удалить из него все операторы, а потом поместим в него три версии одного запроса:

```
tkyte@TKYTE816> connect tkyte/tkyte

tkyte@TKYTE816> alter system flush shared_pool;

System altered.

tkyte@TKYTE816> select * from t where x = 5;

no rows selected

tkyte@TKYTE816> alter session set optimizer_goal=first_rows;

Session altered.

tkyte@TKYTE816> select * from t where x = 5;

no rows selected


tkyte@TKYTE816> connect scott/tiger

scott@TKYTE816> select * from t where x = 5;

no rows selected

scott@TKYTE816> connect tkyte/tkyte

tkyte@TKYTE816> select sql_text, version count
  2     from v$sqlarea
  3    where sql_text like 'select * from t where x = 5%'
  4   /
SQL TEXT                  VERSION COUNT

select*fromtwherex=5


tkyte@TKYTE816> select loaded_versions, optimizer_mode,
  2           parsing_user_id, parsing_schema_id
  3     from v$sql
  4    where sql_text like 'select * from t where x = 5%'
  5   /
```

```
LOADED_VERSIONS  OPTIMIZER_  PARSING_USER_ID  PARSING_SCHEMA_ID
---------------  ----------- ---------------- ------------------
              1  CHOOSE                    69                 69
              1  FIRST_ROWS                69                 69
              1  CHOOSE                    54                 54
```

Это объясняет, почему в разделяемом пуле оказалось несколько версий. Первые две версии появились потому, что, хотя запросы и разбирались одним пользователем, но разбор выполнялся в различных средах. Первый раз оптимизатор работал в режиме CHOOSE, второй раз — в режиме FIRST ROWS. Поскольку другой режим работы оптимизатора может привести к выбору другого плана выполнения запроса, необходимо хранить две версии запроса. Третья строка появилась потому, что запрос, хоть и совпадает по тексту, совсем другой. Этот запрос выбирает данные из таблицы SCOTT.T, а не из TKYTE.T; это принципиально другой запрос.

Большого количества версий одного оператора надо избегать по той же причине, что и использовать связываемые переменные или избегать мягких разборов, — чтобы не делать лишних действий. Иногда несколько версий одного оператора неизбежны, в частности, если эти SQL-операторы выполняются от имени разных учетных записей и применяются к разным таблицам, как в рассмотренном случае с таблицами TKYTE.T и SCOTT.T. Других же случаев, когда хранение нескольких версий связано с выполнением операторов в различных средах, необходимо по возможности избегать.

В рассмотренном случае версии оператора выполнялись от имени 21 учетной записи и применялись к разным таблицам.

```
Instance Activity Stats for DB: 0RA8I  Instance: ora8i  Snaps: 1 -3

Statistic                          Total    per Second    per Trans
---------------------------------------------------------------------
CPU used by this session      14,196,226      18,630.2      41,876.8

parse count (hard)                   127           0.2           0.4
parse count (total)               12,530          16.4          37.0
parse time cpu                       203           0.3           0.6
parse time elapsed                   231           0.3           0.7

sorts (disk)                           9           0.0           0.0
sorts (memory)                     2,242           2.9           6.6
sorts (rows)                     340,568         446.9       1,004.6
```

Эта часть отчета, Instance Activity Stats, содержит много точных значений. Многие из них мы уже видели — они использовались для вычисления коэффициентов и статистических показателей в начале отчета. Например, по значениям parse count (hard) и (total) можно получить:

```
tkyte@TKYTE816> select round( 100 *(1-127/12530) ,2 ) from dual;

 ROUND(100*(1-127/12530),2)
---------------------------
                      98.99
```

что в точности соответствует значению параметра Soft Parse %, представленного в начале отчета. Эти точные данные использовались для вычисления многих представленных ранее показателей.

```
Tablespace IO Stats for DB: ORA8I  Instance: ora8i  Snaps: 1 -3
->ordered by IOs (Reads + Writes) desc
Tablespace
------------------------------
                 Av       Av      Av                Av       Buffer  Av Buf
       Reads  Reads/s  Rd(ms)  Blks/Rd   Writes  Writes/s   Waits   Wt(ms)

TEMP
       1,221        2     0.0      2.3       628         1       0      0.0

File 10 Stats for DB: ORA8I  Instance: ora8i  Snaps: 1 -3
->ordered by Tablespace, File

Tablespace     Filename

                 Av       Av      Av                Av         Buffer Av Buf
       Reads  Reads/s Rd(ms)  Blks/Rd   Writes  Writes/s    Waits  Wt(ms)

DRSYS                          /d02/oradata/ora8i/drsys01.dbf
          14         0 7 . 9      2.4         0         0        0
```

Представленные выше фрагменты отчета связаны с вводом/выводом. Надо добиваться равномерного распределения операций чтения и записи по устройствам. По этому фрагменту отчета можно определить "горячие" файлы. Понимая, как читаются и записываются данные, администратор базы данных сможет добиться повышения производительности за счет более равномерного распределения ввода/вывода по дискам.

```
Buffer Pool Statistics for DB: ORA8I  Instance: ora8i  Snaps: 1 -3
-> Pools   D: default pool,  K: keep pool,  R: recycle pool

                          Free     Write      Buffer
     Buffer  Consistent   Physical Physical   Buffer Complete     Busy
P      Gets        Gets      Reads   Writes    Waits    Waits     Waits
-- -------- ---------- ---------- --------- --------- -------- ---------
D     9,183     721,865      7,586       118         0        0         0
```

Если используется поддержка нескольких буферных пулов, в этом разделе представляются данные по каждому из них. В нашем случае просто повторяется общая информация, представленная в начале отчета.

```
Rollback Segment Stats for DB: ORA8I  Instance: ora8i  Snaps: 1 -3
->A high value for "Pet Waits" suggests more rollback segments may be required

        Trans Table      Pet  Undo Bytes
RBS No      Gets        Waits    Written    Wraps    Shrinks Extends

      0       5.0         0.00          0        0          0       0
      1     866.0         0.00    447,312        1          0       0
```

```
Rollback Segment Storage for DB: ORA8I  Instance: ora8i  Snaps: 1 -3
->Optimal Size should be larger than Avg Active

RBS No    Segment Size   Avg Active  Optimal Size  Maximum Size

     0         663,552        7,372                     663,552
     1      26,206,208      526,774                  26,206,208
     2      26,206,208      649,805                  26,206,208
```

В этом разделе представлена информация об использовании сегментов отката. В этом случае также имеет смысл добиваться равномерного распределения нагрузки по сегментам отката (за исключением, конечно, сегмента отката в табличном пространстве SYSTEM). Кроме того, в заголовке раздела представлены наиболее существенные соображения, которые необходимо учитывать при анализе этой информации. Обратите внимание на совет, что значение Optimal должно быть больше, чем Avg Active, если установка оптимального размера сегментов отката вообще используется (представленный отчет показывает, что в этой базе данных оптимальный размер сегментов отката не задан). Поскольку определение размера сегментов отката и распределение ввода/вывода по ним — это задача администратора базы данных, мы переходим к следующему разделу:

```
Latch Activity for DB: ORA8I  Instance: ora8i  Snaps: 1 -3
->"Get Requests", "Pet Get Miss" and "Avg Sips/Miss" are statistics for
willing-to-wait latch get requests
->"NoWait Requests", "Pet NoWait Miss" are for no-wait latch get requests
->"Pct Misses" for both should be very close to 0.0

                                             Pct    Avg            Pct
                                    Get      Get   Slps   NoWait NoWait
Latch Name                       Requests   Miss  /Miss Requests   Miss
                                 ---------------------------------------
Active checkpoint queue latch         271    0.0                0

virtual circuit queues                 37    0.0                0

Latch Sleep breakdown for DB: 0RA8I  Instance: ora8i  Snaps: 1 -3
-> ordered by misses desc

                                 Get                          Spin &
Latch Name                    Requests    Misses    Sleeps  Sleeps 1->4

library cache                  202,907        82        12  72/8/2/0/0
cache buffers chains         2,082,767        26         1  25/1/0/0/0

Latch Miss Sources for DB: ORA8I  Instance: ora8i  Snaps: 1 -3
-> only latches with sleeps are shown
-> ordered by name, sleeps desc

                                                   NoWait            Waiter
Latch Name            Where                        Misses   Sleeps   Sleeps

cachebufferschainsk c l b g t c r :kslbegin                     0        1 1
```

```
library cache          kglic                          0       7       0
library cache          kglhdgn: child:                0       3       1
library cache          kglget: child: KGLDSBYD        0       2       0

Child Latch Statistics DB: 0RA8I  Instance: ora8i  Snaps: 1 -3
-> only latches with sleeps are shown
-> ordered by name, gets desc

                     Child        Get
Latch Name             Num   Requests    Hisses  Sleeps
-------------------------------------------------------

    Spin &
 Sleeps 1->4
-----------

cache buffers chains  930     93,800        21       1
20/1/0/0/0
library cache           2     48,412        34       6
29/4/1/0/0
library cache           1     42,069        10       3
8/1/1/0/0
library cache           5     37,334        10       2
8/2/0/0/0
library cache           4     36,007        13       1
12/1/0/0/0
```

Как было описано в главе 3, защелки — это простые средства обеспечения последовательного доступа в СУБД Oracle. Защелка всегда либо устанавливается, либо нет, в отличие от очередей, где не имеющий возможности установить блокировку процесс "засыпает", пока блокировка не будет снята другим процессом. При использовании защелки запрашивающий процесс сразу определяет, установлена она или нет. Если защелка не установлена, запрашивающий процесс некоторое время "крутится" (используя ресурсы процессора), пытаясь установить защелку еще раз. Если не получается, он "засыпает" на некоторое время и пытается установить ее еще раз. В представленных выше отчетах отражена информация об этих действиях. Например, видно, что защелку библиотечного кэша не удалось установить 82 раза из 202907 попыток. Далее, 72 из этих 82 защелок были успешно установлены при следующей попытке, 8 — при второй и 2 — при третей. Показатель установки с первой попытки в системе был близок к 100 процентам (почти 100 процентов устанавливавшихся защелок были успешно установлены сразу же), так что тут проблем нет. В системе, не использующей связываемые переменные или слишком часто разбирающей запросы, вы увидите многочисленные конфликты при установке защелок в библиотечном кэше. Еще по этому фрагменту отчета можно понять, что около 4,5 процентов (93800/2082767) запросов защелки на цепочки кэш-буферов приходилось на одну дочернюю защелку из 930. Это, вероятно, означает, что в системе имеется "горячий" блок, к которому одновременно пытается обратиться несколько процессов. Им всем нужна защелка, чтобы обратиться к этому блоку, и это приводит к конфликтам. С этой проблемой надо разобраться. Отчет о защелках помогает найти подобные конфликты. Для их снятия придется вернуться к настройке на уровне **приложений.** Конфликты при установке защелок — это симптом, а не причина проблемы.

Чтобы избавиться от симптома, надо установить причину. К сожалению, нельзя получить список рекомендаций вида "если имеются конфликты при установке такой-то защелки, надо делать то-то" (если бы все было так просто!). Если установлено наличие конфликтов при установке защелок, надо вернутся к приложению и определить, при обращении к какому ресурсу происходит конфликт.

```
Dictionary Cache Stats for DB: 0RA8I  Instance: ora8i  Snaps: 1 -3
->"Pct Misses"   should be very low (< 2% in most cases)
->"Cache Usage"  is the number of cache entries being used
->"Pct SGA"      is the ratio of usage to allocated size for that cache
```

Cache	Get Requests	Pct Miss	Scan Requests	Pct Miss	Mod Req	Final Usage	Pct SGA
dc_constraints	0		0		0	227	99
dc database links	9	0.0	0		0	7	88
dc files	0		0		0	69	88
dc free_extents	747	55.0	336	0.0	672	90	98
dc_global_oids	14	0.0	0		0	95	86
dc_histogram_data	0		0		0	0	0
dc_histogram_data_valu	0		0		0	0	0
dc_histogram_defs	94	21.3	0		0	1,902	100
dc_object_ids	190	0.0	0		0	2,392	100
dc_objects	**345**	2.9	0		0	6,092	100
dc outlines	0		0		0	0	0
dc_profiles	132	0.0	0		0	2	33
dc_rollback_segments	192	0.0	0		0	33	77
dc segments	637	0.6	0		340	3,028	100
dc sequence grants	6	0.0	0		0	6	6
dc_sequences	8	0.0	0		5	23	82
dc synonyms	28	10.7	0		0	96	96
dc tablespace quotas	0		0		0	14	12
dc_tablespaces	1,033	0.0	0		0	100	94
dc used extents	672	50.0	0		672	5	6
dc_user_grants	1,296	0.2	0		0	756	82
dc_usernames	337	0.9	0		0	1,318	99
dc users	9,892	0.0	0		1	865	100

Это отчет об использовании кэша словаря. Мне он не слишком нравится, поскольку я практически не могу повлиять на выдаваемые в нем числа. Кэш словаря целиком управляется сервером Oracle, и мы не можем изменить размеры его компонентов. Можно задавать только размер разделяемого пула, и если размер этот задан корректно, сервер сам о себе позаботится. Поскольку у меня разделяемый пул используется на 75 процентов, его размер вполне достаточен. Если бы разделяемый пул был "заполнен" и процент попаданий был низким, увеличение разделяемого пула позволило бы увеличить этот процент.

```
Library Cache Activity for DB: ORA8I  Instance: ora8i  Snaps: 1 -3
->"Pct Misses"   should be very low
```

Namespace	Get Requests	Pct Miss	Pin Requests	Pet Miss	Reloads	Invali- dations
BODY	5,018	0.0	5,018	0.0	0	0
CLUSTER	0		0		0	0
INDEX	1	0.0	1	0.0	0	0
OBJECT	0		0		0	0
PIPE	765	0.0	765	0.0	0	0
SQL AREA	1,283	6.9	38,321	0.6	39	0
TABLE/PROCEDURE	3,005	0.3	11,488	0.6	1	0
TRIGGER	21	0.0	21	0.0	0	0

Здесь выдается информация о коэффициентах попадания в библиотечный кэш по объектам. В хорошо настроенной системе значение Pct Misses близко к нулю. В системе, используемой для разработки, или в той, где объекты часто создаются и удаляются, некоторые показатели будут иметь большие значения, например, столбец Invalidations. Установка соответствующего размера разделяемого пула и сведение к минимуму количество жестких разборов за счет использования связываемых переменных — вот путь получения хороших значений в данном разделе.

```
SGA Memory Summary for DB: 0RA8I  Instance: ora8i  Snaps: 1 -3

SGA regions          Size in Bytes

Database Buffers       134,217,728
Fixed Size                  69,616
Redo Buffers               532,480
Variable Size          130,187,264

                       265,007,088

SGA breakdown difference for DB: ORA8I  Instance: ora8i  Snaps: 1 -3

Pool        Name                         Begin value   End value Difference
---------------------------------------------------------------------------
Java pool   free memory                   17,838,080  17,838,080          0
java pool   memory in use                  3,133,440   3,133,440          0
shared pool KGFF heap                         54,128      54,128          0
shared pool KGK heap                           5,840       5,840          0
shared pool KQLS heap                      3,253,784   3,231,844    -21,940
shared pool PL/SQL DIANA                   4,436,044   4,413,960    -22,084
shared pool PL/SQL MPCODE                 15,378,764  15,546,652    167,888
shared pool PLS non-lib hp                     2,096       2,096          0
shared pool State objects                    291,304     291,304          0
shared pool VIRTUAL CIRCUITS                 484,632     484,632          0
shared pool db block buffers               2,228,224   2,228,224          0
shared pool db_block_hash_buckets            393,240     393,240          0
shared pool dictionary cache               6,547,348   6,586,032     38,684
shared pool event statistics per ses       1,017,600   1,017,600          0
shared pool fixed allocation callbac             960         960          0
shared pool free memory                   27,266,500  27,013,928   -252,572
shared pool joxlod: in ehe                    71,344      71,344          0
```

```
shared pool   joxs heap init                244            244          0
shared pool   library cache           28,105,460     28,168,952     63,492
shared pool   message pool freequeue     231,152        231,152          0
shared pool   miscellaneous            1,788,284      1,800,404     12,120
shared pool   pl/sql source               42,536         42,536          0
shared pool   processes                   153,600        153,600          0
shared pool   sessions                    633,600        633,600          0
shared pool   sql area                 16,377,404     16,390,124     12,720
shared pool   table columns               45,264         45,936        672
shared pool   table definiti              11,984         12,944        960
shared pool   transactions                294,360        294,360          0
shared pool   trigger defini               8,216          8,216          0
shared pool   trigger inform               5,004          5,064         60
shared pool   type object de              48,040         48,040          0
shared pool   view columns d               1,072          1,072          0
              db_block_buffers       134,217,728    134,217,728          0
              fixed_sga                   69,616         69,616          0
              log_buffer                 512,000        512,000          0
```

В этой части отчета использование разделяемого пула показано более детально. Можно увидеть, как со временем меняется использование памяти каждым из компонентов: некоторые освобождают память, другие — захватывают. Мне эта часть отчета кажется полезной, поскольку объясняет результаты, представленные в других частях. Например, я получил серию отчетов StatsPack для анализа. Они показывали относительно стабильные количества жестких и мягких разборов и вдруг, абсолютно неожиданно, количество жестких разборов превзошло все мыслимые пределы примерно на час, а затем вернулось к обычному уровню. По этому разделу отчета я смог определить, что одновременно с ростом количества жестких разборов существенно (на много десятков мегабайт) уменьшилось использование памяти в области SQL разделяемого пула. Пытаясь понять это, я спросил: "Никто не освобождал разделяемый пул?" и получил ответ: "Конечно, да". Это было стандартной процедурой в ходе работы: каждые шесть часов очищать разделяемый пул. Зачем? Никто не знал — просто всегда так делали. Для этого даже было создано специальное задание. Отключение этого задания решило проблему периодического снижения производительности, которое было вызвано сбросом содержимого разделяемого пула (и одновременно всех планов выполнения запросов, накопленных за шесть часов).

```
init.ora Parameters for DB: ORA8I  Instance: ora8i  Snaps: 1 -3

                                                    End value
Parameter Name           Begin value                (if different)

background_dump_dest     /export/home/ora816/admin/ora8i/b
compatible               8.1.0,  8.1.6.0.0
control_files            /d01/oradata/ora8i/control01.ctl,
core_dump_dest           /export/horae/ora816/admin/ora8i/c
db_block_buffers         16384

End of Report
```

В конце отчета приведены параметры инициализации экземпляра, значения которых отличаются от стандартных. Это, наряду с подробным отчетом об использовании памяти в разделяемом пуле, помогает определить причины тех или иных событий. Можно быстро понять, какие параметры установлены явно и как это повлияло на работу системы.

Пакет StatsPack — отличное средство как для производственной среды, так и для среды разработки. Оно поможет определить общий уровень "здоровья" базы данных, а также выявить узкие места. Я настоятельно рекомендую постоянно его использовать при эксплуатации системы. Есть планы по добавлению дополнительных средств анализа информации, получаемой с помощью StatsPack, в частности для анализа тенденций, чтобы можно было не только видеть изменения, произошедшие между двумя точками, но и тенденцию изменений за какой-то период времени.

Понимание содержимого отчета — это первый шаг. Дальше нужно выработать план реагирования на получаемые результаты. Нельзя рассматривать одно значение или показатель изолированно, поскольку для правильной интерпретации необходимо четкое понимание назначения и устройства системы. В рассмотренном ранее отчете, если бы я не знал, что пакетное задание **sync_users** интенсивно работает с удаленной базой данных и что это — фоновый процесс, то мог бы подумать, что столкнулся с проблемой в системе. Но я не стал паниковать, зная, что в фоновом режиме работает процедура **sync_users**. То, что она некоторое время ждет доступа к удаленной базе данных — вполне приемлемо и не выходит за рамки ограничений системы. Отчет показал, что в системе есть приложение, неоптимально использующее связываемые переменные — слишком часто выполняется жесткий разбор. Для определения причин этой проблемы я бы использовал другие инструментальные средства, представленные ранее.

Представления V$

В этом разделе я хочу рассмотреть основные представления V$, необходимые при настройке приложений. Этих представлений — более 180, и хотя все они называются "представления динамической производительности", не все они связаны с производительностью. Мы рассмотрим те, которые я использую постоянно. Есть и другие представления, используемые, например, для контроля и настройки многопоточного сервера; я их описывать не буду. Все эти представления детально описаны в руководстве *Oracle8i Reference*. Я не собираюсь воспроизводить эти описания, хочу лишь обратить внимание на те представления, о которых обязательно надо знать.

Я кратко опишу эти представления, чтобы вы знали об их существовании и назначении. Многие из них уже использовались в примерах. Другие будут представлены впервые. При необходимости будет даваться небольшой пример использования соответствующего представления.

Представление V$EVENT_NAME

В этой главе неоднократно упоминались многие события. Было показано, что событие имеет имя и до трех параметров: p1, p2 и p3. Если необходимо узнать интерпрета-

цию параметров p1, p2 и p3 для события, можно обратиться к документации либо выполнить запрос к представлению **V$EVENT_NAME.** Например, в этой главе рассматривались события **latch free** и **enqueue.** Выполнив запросы к этому представлению:

```
tkyte@TKYTE816>  select * from v$event_name where name = 'latch free'
  2  /

    EVENT# NAME            PARAMETER1   PARAMETER2   PARAMETER3

         2 latch free         address       number        tries

tkyte@TKYTE816>  select * from v$event_name where name = 'enqueue'
  2  /

    EVENT* NAME            PARAMETER1   PARAMETER2   PARAMETER3

        11 enqueue           name|mode          id1          id2
```

можно понять назначение этих параметров, особенно если мы про них уже читали в документации, и надо просто вспомнить, что именно и в каком порядке выдается.

Представления V$FILESTAT и V$TEMPSTAT

Представления **V$FILESTAT** и **V$TEMPSTAT** позволяют быстро оценить объем ввода/вывода в системе, а также количество времени, потраченное сервером Oracle на чтение и запись любого файла. Оценку этих параметров можно получить с помощью средств пакета StatsPack либо сравнив значения в этих представлениях через определенный период времени.

Представление V$LOCK

Это представление я уже несколько раз использовал в главе 3. Оно позволяет понять, кто и кого блокирует. Помните, сервер Oracle не хранит блокировки уровня строк отдельно отданных, так что не ищите их в этом представлении. В нем можно обнаружить, кто установил блокировки TM (DML Enqueue) на таблицы; поэтому можно понять, что сеанс 'x,y' заблокировал некоторые строки в таблице, но нельзя понять **какие именно.**

Представление V$MYSTAT

Это представление содержит статистическую информацию только о **запрашивающем сеансе.** Схема, обращающаяся к этому представлению, должна иметь непосредственный доступ к базовым объектам **V_$STATNAME** и **V_$MYSTAT,** например:

```
sys@TKYTE816> grant select on v_$statname to tkyte;
    Grant succeeded.

sys@TKYTE816> grant select on v_$mystat to tkyte;
    Grant succeeded.
```

Обратите внимание, что использовано имя **V_$STATNAME,** а не **V$STATNAME.** Дело в том, что **V$STATNAME** — это всего лишь общедоступный синоним для представления **V_$STATNAME.**

Это представление содержит номер статистического показателя, код, а не имя отслеживаемого события. Я обычно создаю следующее представление:

```
ops$tkyte@ORA8I.WORLD> create view my_stats
  2  as
  3  select a.name, b.value
  4    from v$statname a, v$mystat b
  5   where a.statistic# = b.statistic#
  6  /

view created.

ops$tkyte@ORA8I.WORLD> SELECT * FROM MY_STATS WHERE VALUE > 0;

NAME                             VALUE

logons cumulative                    1
logons current                       1
opened cursors cumulative          160
opened cursors current               1
```

в используемых системах, чтобы упростить запросы. После создания этого представления можно делать запросы, выдающие информацию о сеансе в стиле пакета StatsPack. Например, вот как можно вычислить немаловажный параметр **Soft Parse Ratio** — процент мягких разборов:

```
ops$tkyte@ORA8I.WORLD> select round(100 *
  2              (1-max(decode(name,'parse count (hard)',value,null))/
  3                 max(decode(name,'parse count (total)',value,null))), 2
  4                                 ) "Soft Parse Ratio"
  5     from my_stats
  6  /

Soft Parse Ratio

            84.03
```

Если создать набор подобных запросов и вызывать их в триггере на системное событие **logoff** или встроить непосредственно в приложение, можно будет контролировать производительность (сколько транзакций зафиксировано, сколько откачено и т.д.) каждого сеанса и приложения.

Представление V$OPEN_CURSOR

Это представление содержит список открытых курсоров для всех сеансов. Это очень полезно для выявления "утечек" курсоров и определения, какие именно операторы SQL выполняются сеансом. Сервер Oracle оставляет курсоры в кэше даже после их явного закрытия, так что не удивляйтесь, если обнаружите в результатах закрытые курсоры (это вполне возможно). Например, в том же сеансе SQL*Plus, в котором выше вычислялся коэффициент **Soft Parse Ratio,** я обнаружил:

```
ops$tkyte@ORA8I.WORLD> select * from v$open_cursor
  2  where sid = (select sid from v$mystat where rownum = 1);

SADDR    SID USER_NAME ADDRESS  HASH_VALUE SQL_TEXT

8C1706A0  92 OPS$TKYTE 8AD80D18  607327990 BEGIN DBMS_OUTPUT.DISABLE;
                                           END;

8C1706A0  92 OPS$TKYTE 8AD6BB54  130268528 select lower(user) || '@' ||
                                           decode(global_name,
                                           'ORACLE8.WO

8C1706A0  92 OPS$TKYTE 8AD8EDB4  230633120 select round(100 *
                                           (1-max(decode(name,'parse
                                           count (hard

8C1706A0  92 OPS$TKYTE 8AD7DEC0 1592329314 SELECT
                                           ATTRIBUTE,SCOPE,NUMERIC_VALUE,
                                           CHAR_VALUE,DATE_VALUE F

8C1706A0  92 OPS$TKYTE 8E16AC30 3347301380 select round( 100 *
                                           (1-max(decode(name,'parse
                                           count (hard)',

8C1706A0  92 OPS$TKYTE 8AD7AD70 1280991272 SELECT CHAR_VALUE FROM
                                           SYSTEM.PRODHCT_PRIVS WHERE
                                           (UPPER('

8C1706A0  92 OPS$TKYTE 8AD62080 1585371720 BEGIN
                                           DBMS_OUTPUT.ENABLE(1000000);
                                           END;

8C1706A0  92 OPS$TKYTE 8AD816B8 3441224864 SELECT USER FROM DUAL
8C1706A0  92 OPS$TKYTE 8ADF4D3C 1948987396 SELECT DECODE('A','A','1','2')
                                           FROM DUAL

8C1706A0  92 OPS$TKYTE 89D30A18 2728523820 select round(100 *
                                           (1-max(decode(name,'parse
                                           count (hard

8C1706A0  92 OPS$TKYTE 8865AB90 3507933882 select * from v$open_cursor
                                           where sid = ( select sid from
                                           v$
8C1706AO  92 OPS$TKYTE 8AD637B0 242587281  commit
8C1706A0  92 OPS$TKYTE 8AD70660 3759542639 BEGIN
                                           DBMS_APPLICATION_INFO. SET_MODU
                                           LE(:1,NULL); END;

13 rows selected.
```

Как видите, есть некоторое количество открытых курсоров. Однако:

```
ops$tkyte@ORA8I.WORLD> select * from my_stats where name = 'opened cursors
   current';

NAME                           VALUE

opened cursors current            1
```

На самом деле открыт один курсор (причем это именно тот курсор, с помощью которого выбирается информация об открытых курсорах). Сервер Oracle держит другие курсоры в кэше на случай повторного выполнения запросов.

Представление V$PARAMETER

Представление V$PARAMETER помогает получить значения различных установок, связанных с настройкой, например размер блока, размер области сортировки и т.д. Они имеют отношение к настройке, поскольку многие из этих параметров инициализации непосредственно влияют на производительность.

Представление V$SESSION

Представление V$SESSION содержит строку для каждого сеанса. Как и в случае рассмотренного ранее представления V$STATNAME, для использования этого представления администратор базы данных должен предоставить вам соответствующие привилегии:

```
sys@TKYTE816> grant select on v_$session to tkyte;
    Grant succeeded.
```

Чтобы найти строку для текущего сеанса, можно выполнить следующий запрос:

```
ops$tkyte@ORA8I.WORLD> select * from v$session
  2   where sid = (select aid from v$mystat where rownum = 1)
  3   /
```

Я обычно использую это представление, чтобы понять, что же еще происходит в базе данных. Например, я часто использую сценарий **showsql,** показывающий мне список сеансов с информацией о состоянии сеанса (активен или нет), выполняемом модуле, действии и параметрах **client_info**, и, наконец, о выполняемом SQL-операторе для активных сеансов.

Поля **MODULE**, **ACTION** и **CLIENT_INFO** может устанавливать разработчик приложений с помощью вызовов соответствующих процедур пакета **DBMS_APPLICATION_INFO** (подробное описание этого пакета представлено в Приложении А). Я рекомендую устанавливать эти поля в каждом создаваемом приложении. Это может сэкономить много времени при попытках определить, какое приложение выполняется в том или ином сеансе (если эта информация есть в представлении V$, ответ очевиден).

Мой сценарий **showsql** имеет вид:

```
column username format a15 word_wrapped
column module format a15 word_wrapped
column action format a15 word_wrapped
column client_info format a15 word_wrapped
column status format a10
column sid_serial format a15
set feedback off
set serveroutput on
```

```
select username, ''''||sid| |','||serial#||'''' sid_serial, status , modu
action, client_info
from v$session
where username is not null


column username format a20
column sql_text format a55 word wrapped

set serveroutput on size 1000000
declare
     x number;
procedure p  (p_str in varchar2)
is
    l_str    long := p_str;
begin
    loop
        exit when l_str is null;
        dbms_output.put_line(substr(l_str,  1,  250));
        l_str := substr(l_str, 251);
    end loop;
end;
begin
     for x in
     (select username||'('||sid||','|| serial#||
                    ') ospid = '  ||  process ||
                    ' program =  '  ||  program username,
                to_char(LOGON_TIME,' Day HH24:MI') logon_time,
                to_char(sysdate,' Day HH24:MI') current_time,
                sql_address, LAST_CALL_ET
          from v$session
         where status =  'ACTIVE'
           and rawtohex(sql_address) <> '00'
           and username is not null order by last_call_et)
     loop
          dbms_output.put_line('--------------------') ;
          dbms_output.put_line(x.username ) ;
          dbms_output.put_line(x.logon_time  ||  '  ' ||
                               x.current_time||
                                ' last at =  ' ||
                               x.IAST_CALL_ET);
          for y in  (select sql_text
                          from v$sqltext_with_newlines
                         where address = x.sql_address
                           order by piece)
          loop
               p(y.sql_text) ;
          end loop;
     end loop;
end;
```

и выдает такие результаты:

```
ops$tkyte@ORA8I.WORLD>  @showsql

USERNAME        SID_SERIAL   STATUS    MODULE          ACTION   CLIENT_INFO

OPS$TKYTE       '30,23483'   ACTIVE    01@  showsql.sql
CTXSYS          '56,32'      ACTIVE

OPS$TKYTE(30,23483)  ospid = 29832 program = sqlplusSaria  (TNS V1-V3)
Sunday     20:34   Sunday     20:40 last et = 0
SELECT USERNAME || ' ( ' || SID || ',' || SERIAL# || ') ospid
= '  || PROCESS || ' program = ' || PROGRAM USERNAME,TO_CHAR(
LOGON_TIME,' Day HH24:MI') LOGON_TIME,TO_CHAR(SYSDATE,' Day HH24
:MI') CURRENT_TIME,SQL_ADDRESS,LAST_CALL_ET   FROM V$SESSION  WH
ERE STATUS = 'ACTIVE'  AND RAWTOHEX(SQL_ADDRESS) != '00'  AND US
ERNAME IS NOT NULL  ORDER BY LAST_CALL_ET

CTXSYS(56,32)  ospid = 15610 program = ctxsrv@aria  (TNS VI-V3)
Monday     11:52   Sunday     20:40 last et = 20
BEGIN      drilist.get_cmd( :sid, :mbox, :pmask, :cmd_type,:disp_
id, :disp_return_address, :disp_user, :disp_command, :disp_arg1,
:disp_arg2, :disp_arg3, :disp_arg4, :disp_arg5, :disp_arg6, :di
sp_arg7, :disp_arg8, :disp_arg9, :disp_arglO ) ; :error_stack : =
drue.get_stack; exception when dr_def.textile_error then :error_
stack := drue.get_stack; when others than drue.text_on_stack(sql
errm); :error_stack := drue.get_stack; END;
ops$tkyte@ORA8I.WORLD>
```

Как видите, утилита SQL*Plus заполнила столбец MODULE представления V$SESSION именем выполняющегося сценария. Эта информация может очень помочь, особенно если приложения меняют значение в этом столбце, чтобы показать свое текущее состояние.

Представление V$SESSION_EVENT

Мы уже несколько раз использовали это представление. Я часто использую его, чтобы понять, что заставляет процедуру или запрос "ждать" ресурса. Аналогичную информацию можно получить из трассировочного файла при соответствующей установке событий, но это представление облегчает получение текущих времен ожидания событий для сеанса и сравнение их с последующими, полученными после запуска того или иного процесса. Это намного проще, чем выискивать соответствующую информацию в трассировочном фале.

Это представление содержит информацию об ожидании событий для всех сеансов в системе, поэтому позволяет узнать, чего именно ждут другие сеансы, а не только запрашивающий. Аналогично включению трассировки для другого сеанса с помощью пакета **DBMS_SYSTEM**, можно использовать представление V$SESSION_EVENT для слежения за ожиданием событий другими.

Представление V$SESSION_LONGOPS

Это представление будет подробно рассмотрено в Приложении А. Оно используется продолжительными процессами, такими как создание индексов, резервное копирование и восстановление и любыми другими, которые, "по мнению" стоимостного оптимизатора, потребуют более шести секунд для информирования о ходе работы. Приложения также могут использовать это представление с помощью пакета DBMS_APPLICATION_INFO. Если создается длительно работающий процесс или задание, в нем можно выполнять вызовы процедур пакета DBMS_APPLICATION_INFO для информирования других сеансов о ходе работы. В этом случае легко контролировать работу задания и определять, зависло оно или просто для решения задачи нужно время.

Представление V$SESSION_WAIT

Это представление содержит информацию обо всех сеансах, находящихся в состоянии ожидания, и о времени ожидания. Оно обычно используется для слежения за предположительно зависшими или слишком медленно работающими приложениями.

Представление V$SESSTAT

Представление **V$SESSTAT** аналогично **V$MYSTAT**, но содержит статистическую информацию обо всех сеансах, а не только о запрашивающем. Оно используется для контроля сеансов, работа которых вас интересует.

Например, это представление можно использовать для контроля процента мягких разборов в приложении стороннего производителя, установленного на том же сервере. Это часто приходится делать при росте количества жестких разборов в ранее хорошо настроенной системе. Контролируя процент мягких разборов только этого нового приложения, можно быстро определить, не оно ли стало причиной появления в системе множества уникальных SQL-операторов, не использующих связываемые переменные.

Представление V$SESS_IO

Оно позволяет понять, какой объем ввода/вывода выполнил текущий (или любой другой) сеанс. Я использую это представление аналогично представлениям **V$MYSTAT** и **V$SESSION_EVENT**. Я делаю моментальный снимок, выполняю ряд действий, а затем определяю "разницу" между двумя моментами времени. Она показывает, какой объем ввода/вывода был выполнен в ходе рассматриваемых действий. Эту информацию можно получить и из отчета **TKPROF**, но в запросах легко выполнять подсчеты и агрегировать результаты. Утилита **TKPROF** будет показывать, сколько операций ввода/вывода потребовал каждый оператор. Запросы же к представлению V$SESS_IO позволяют выполнять произвольный набор операторов и получать статистические данные, касающиеся ввода/вывода для всего набора в целом.

Представления V$SQL и V$SQLAREA

Эти представления содержат разобранные и хранящиеся в разделяемом пуле SQL-операторы. Оба эти представления уже использовались в нескольких главах.

Представление **V$SQLAREA** — обобщающее. Оно будет содержать по одной строке для каждого SQL-запроса. Столбец **VERSION_COUNT** показывает, сколько строк содержится в представлении **V$SQL** для соответствующего запроса. Старайтесь избегать запросов к этому представлению; обращайтесь непосредственно к представлению **V$SQL**. Получение информации из **V$SQLAREA** может потребовать слишком много ресурсов, особенно в загруженной системе.

Представления **V$SQLAREA** и **V$SQL** позволяют увидеть, какие SQL-операторы выполняются в системе, сколько раз каждый оператор выполняется и разбирается, сколько логических и физических операций ввода/вывода он делает и т.д. Эти представления также позволяют находить SQL-операторы, не использующие связываемых переменных.

Представление V$STATNAME

Представление V$STATNAME содержит имена всех статистических показателей. Оно используется в соединениях с представлениями **V$MYSTAT** и **V$SESSTAT** для преобразования номера показателя в понятное имя.

Представление V$SYSSTAT

Тогда как представление **V$SESSTAT** содержит статистическую информацию по сеансам, в **V$SYSSTAT** она накапливается для экземпляра в целом. Сеансы начинаются и завершаются, соответствующие записи добавляются и удаляются в базовые таблицы представления **V$SESSTAT**, а данные в представлении **V$SYSSTAT** существуют до тех пор, пока сервер не будет остановлен. Именно эту информацию использует пакет StatsPack для вычисления большинства выдаваемых показателей.

Представление V$SYSTEM_EVENT

Это представление для ожиданий событий играет ту же роль, что и представление **V$SYSSTAT** для статистических показателей. Оно содержит информацию об ожидании событий на уровне экземпляра. Эту информацию пакет StatsPack также использует при вычислении выдаваемых показателей.

Резюме

Настройка, выполняемая постфактум, — это немного удачи и много поисков. Если же настройка выполняется в ходе разработки, делать ее легко и просто. Я всегда предпочитаю простые и понятные решения, особенно если другие варианты подразумевают "удачу". Настройка постфактум — одно из самых сложных дел. Необходимо разобраться, почему система работает медленно, где именно происходит замедление и как заста-

вить ее работать быстрее, не переделывая полностью заново. Именно последнее требование и делает настройку постфактум настолько сложной.

Если думать о производительности на всех этапах жизненного цикла приложения, окажется, что это — наука, а не искусство. Все могут это делать, если это регламентировано. Для этого необходимо определить показатели, по которым будет проверяться производительность приложений. Необходимо снабдить код средствами контроля и отладки, чтобы можно было определить, где происходит замедление. Включение таких средств — крайне важно; сам сервер обильно снабжен такими средствами, как было показано в этой главе. Эти средства отладки можно использовать для доказательства того, что не СУБД является причиной медленной работы, а какая-то другая система. После этого, если остальной код приложения не снабжен средствами контроля и отладки, придется определять причину проблем наугад. Но и для кода, выполняемого в СУБД, средства контроля и отладки на уровне приложения весьма полезны при выявлении причины медленной работы.

Да, если вы еще не запомнили: **использование связываемых переменных — крайне важно. Я** встречал бессчетное количество систем, обреченных на остановку лишь потому, что разработчикам показалось удобнее использовать во всех запросах конкатенацию строк вместо подстановки переменных. Эти системы не работали. Правильное решение в данном случае — использовать связываемые переменные. Не полагайтесь на фокусы типа установки параметра **CURSOR_SHARING,** поскольку с ними тоже связаны определенные проблемы.

11

Стабилизация плана оптимизатора

Сервер Oracle8i позволяет разработчику сохранить набор "подсказок серверу", описывающих, как выполнять определенные SQL-операторы в базе данных. Эта возможность называется *стабилизацией плана оптимизатора* **(Optimizer Plan Stability)** и реализуется с помощью хранимого шаблона плана выполнения запроса, аналогичного шаблону верстки книги. В этой главе мы подробно рассмотрим эту возможность, в том числе:

- В каких случаях при разработке приложений может понадобиться стабилизация плана оптимизатора, и какие сценарии работы должны при этом использоваться.
- Альтернативные варианты использования этой возможности, не предполагавшиеся разработчиками сервера.
- Стабилизация плана оптимизатора и управление хранимыми шаблонами планов в базе данных как с помощью операторов ЯОД, так и с помощью подпрограмм пакета **OUTLN_PKG.**
- Существенные нюансы, включая чувствительность к регистру символов, проблемы с оператором **ALTER SESSION,** раскрытием условий **OR** и производительностью.
- Ошибки, с которыми можно столкнуться, в том числе отсутствие опций в операторе **ALTER OUTLINE** или наличие шаблона плана с данным именем, и способы их устранения.

Для выполнения примеров в этой главе необходим сервер Oracle8i Release 1 (версия 8.1.5) или более новой версии. Кроме того, это должна быть редакция Oracle8i Enterprise

или Personal Edition, поскольку стабилизация плана оптимизатора в Standard Edition не поддерживается.

Обзор возможностей

Для выполняемого запроса или набора SQL-операторов стабилизация плана оптимизатора позволяет сохранить оптимальный набор подсказок, избавляя от необходимости задавать подсказки в приложении. Это позволяет:

- разработать приложение;
- протестировать и настроить его запросы;
- сохранить соответствующие хорошо настроенные планы выполнения в базе данных для использования оптимизатором в дальнейшем.

Стабилизация плана оптимизатора позволяет защититься от многих изменений используемой базы данных. Существенно изменить планы выполнения запросов могут, в частности, следующие типичные изменения базы данных:

- повторный анализ таблицы после изменения количества данных;
- повторный анализ таблицы после изменения распределения данных;
- повторный анализ таблицы с помощью других методов или параметров;
- изменение различных параметров в файле **init.ora,** влияющих на поведение оптимизатора, например **db_block_buffers;**
- добавление индексов;
- обновление версии ПО Oracle.

Благодаря стабилизации плана запроса, однако, можно сохранить существующие планы выполнения запросов и изолировать приложение от этих изменений.

Следует отметить, что в большинстве случаев желательно, чтобы планы выполнения запросов со временем изменялись в ответ на события из приведенного выше списка. Если распределение данных по столбцу существенно изменяется, оптимизатор соответственно изменяет план выполнения запроса. Если добавлен индекс, оптимизатор выявит и использует его, если это даст преимущество. Стабилизацию плана оптимизатора можно использовать для предотвращения подобных изменений в среде, где изменения должны делаться постепенно, после тщательного тестирования. Например, прежде чем разрешать использование индекса, можно последовательно протестировать запросы, для которых он используется, чтобы убедиться, что добавление индекса не повлияет отрицательно на другие компоненты системы. То же самое справедливо и в отношении изменения параметров инициализации или обновления ПО сервера.

Стабилизация плана оптимизатора реализуется с помощью *подсказок*. Подсказки — это не команды и не правила. Хотя механизм подсказок, лежащий в основе стабилизации плана оптимизатора, мощнее, чем в случае обычных подсказок в тексте запроса, оптимизатор может по ходу работы им и не следовать. Это — палка о двух концах. Кажется, что это — недостаток, но это — полезное свойство. Если в базе данных сделаны такие изменения, что набор подсказок неприменим (например, удален соответствую-

щий индекс), то сервер Oracle будет игнорировать подсказки и генерировать лучший план из возможных.

Продемонстрируем возможности стабилизации плана оптимизатора на простом примере. Ниже представлен один из методов использования хранимого шаблона для запроса. После запоминания шаблона мы сделаем ряд изменений в базе данных (проанализируем таблицу), которые приведут к изменению плана. Наконец, мы увидим, как, включив стабилизацию плана оптимизатора, можно заставить сервер Oracle использовать план, сохраненный первоначально, до сделанных изменений. Сначала создадим копию таблицы SCOTT.EMP и зададим для нее первичный ключ:

```
tkyte@TKYTE816> create  table  amp
  2   as
  3   select ename,  empno from scott.emp group by ename, empno
  4   /

Table created.

tkyte@TKYTE816> alter table emp
  2   add constraint emp_pk
  3   primary key(empno)
  4   /

Table altered.
```

При отсутствии доступа к таблице **EMP** необходимо получить для нее привилегию SELECT. Созданный при добавлении первичного ключа индекс используется в примере; мы генерируем запрос, который его использует. Зададим режим оптимизации CHOOSE.

```
tkyte@TKYTE816> alter session set optimizer_goal=choose
  2   /

Session altered.
```

Это сделано исключительно для согласованности примеров. При отсутствии статистической информации, несомненно, будет вызваться оптимизатор, основанный на правилах. Однако, если установлен другой режим оптимизации, например **FIRST_ROWS,** будет вызываться оптимизатор, основанный на стоимости, и дальнейшие изменения в базе данных могут сказаться на выбираемом плане выполнения запроса. Наконец, вот план выполнения нашего запроса:

```
tkyte@TKYTE816> set autotrace traceonly explain
tkyte@TKYTE816> select empno, ename from amp where empno > 0

Execution Plan

   0       SELECT STATEMENT Optimizer=CHOOSE
   1    0    TABLE ACCESS  (BY INDEX ROWID)  OF  'EMP'
   2    1       INDEX  (RANGE SCAN)  OF  'EMP_PK'  (UNIQUE)
```

Предположим, такой запрос приходит от интерактивного приложения, в котором пользователю желательно получить начальные данные как можно быстрее, и доступ по индексу для этого прекрасно подходит. Нас устраивает этот план выполнения запроса,

и желательно, чтобы он использовался всегда, поэтому мы создадим для запроса соответствующий шаблон. Мы создадим этот шаблон явно (шаблоны можно создавать и неявно, как будет показано в разделе "Метод настройки"):

```
tkyte@TKYTE816> create or replace outline MyOutline
  2  for category mycategory
  3  ON
  4  select empno, ename from emp where empno > 0
  5  /

Outline created.
```

Оператор **CREATE OR REPLACE OUTLINE** создал шаблон запроса и сохранил его в базе данных (где и как он хранится, описано далее в этой главе). Поскольку мы явно создали шаблон, можно задать ему имя **(MYOUTLINE).** Кроме того, мы отнесли этот шаблон запроса к определенной *категории* **(MYCATEGORY).**

Следует отметить, что при выполнении оператора **CREATE OUTLINE** можно получить следующее сообщение об ошибке:

```
select empno, ename from emp where empno > 0
                                 *
ERROR at line 4:
ORA-18005: create any outline privilege is required for this operation
```

Если выдается такое сообщение, необходимо, чтобы администратор базы данных предоставил соответствующему пользователю привилегию **CREATE ANY OUTLINE.** Все привилегии, необходимые для создания и управления шаблонами, описаны в разделе "Как стабилизировать план оптимизатора".

Мы создали шаблон, задающий необходимый план выполнения запроса (просмотр по индексу). Давайте теперь изменим базу данных — просто проанализируем таблицу:

```
tkyte@TKYTE816> analyze table emp compute statistics
  2  /

Table analyzed.
```

Давайте посмотрим, каким теперь будет план выполнения запроса:

```
tkyte@TKYTE816> set autotrace traceonly explain
tkyte@TKYTE816> select empno, ename from emp where empno > 0
  2  /

Execution Plan

   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=14 Bytes=112)
   1    0   TABLE ACCESS (FULL) OF 'EMP' (Cost=1 Card=14 Bytes=112)
```

Вместо использования индекса, как было в режиме оптимизации на основе правил, оптимизатор, основанный на стоимости, срабатывающий благодаря наличию статистической информации, выбирает полный просмотр таблицы. Оптимизатор, основанный на стоимости, выбрал правильный план. В таблице всего 14 строк, и оптимизатор определил, что все они удовлетворяют условию. Однако в нашем приложении все-таки желательно использовать индекс. Чтобы снова использовать предпочтительный план, надо

воспользоваться возможностью стабилизации плана оптимизатора. Для этого достаточно выполнить следующую команду:

```
tkyte@TKYTE816> alter session set use_stored_outlines = mycategory
  2  /

Session altered.
```

Это обеспечивает применение хранимых шаблонов категории MYCATEGORY. Если теперь посмотреть план выполнения запроса:

```
tkyte@TKYTE816> set autotrace traceonly explain
tkyte0TKYTE816> select empno, ename from emp where empno > 0
  2  /

Execution Flan

    0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=14 Bytes=112)
    1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (Cost=2 Card=14
    2    1     INDEX (RANGE SCAN) OF 'EMP_PK' (UNIQUE) (Cost=l Card=14)
```

оказывается, что снова используется исходный план с доступом но индексу. В этом цель стабилизации плана оптимизатора: "заморозить" планы выполнения запросов для хорошо настроенного приложения. Приложение изолируется от изменений планов оптимизатора, происходящих на уровне базы данных (в результате анализа таблиц, выполненного администратором базы данных, изменения параметров инициализации или обновления версии сервера). Как и большинство средств, стабилизация плана оптимизатора — палка о двух концах. То, что внешние изменения не сказываются на приложении, может оказаться как положительным, так и отрицательным. Хорошо это потому, что позволяет добиться предсказуемой производительности в долгосрочной перспективе (поскольку план никогда не изменяется). Однако так можно пропустить новый план, ускоряющий выполнение запроса, и это плохо.

Использование стабилизации плана оптимизатора

В этом разделе мы рассмотрим различные сценарии использования стабилизации плана оптимизатора. Мы будем использовать различные особенности генерации шаблонов плана, не объясняя детали, поскольку создание шаблонов и управление ими подробно описано в следующих разделах.

Метод настройки

Часто спрашивают: "Как задать подсказку для запроса в существующем приложении, не изменяя текста запроса?". Обычно есть доступ только к двоичному коду приложения, так что изменить запрос нельзя, но желательно изменить план его выполнения.

Проблемный запрос известен, более того, изменяя установки сеанса, можно обеспечить требуемую производительность. Если бы можно было добавить в приложение оператор ALTER SESSION (чтобы включить или отключить использование соединения хе-

ширование, например) или добавить в запрос простую подсказку (например, /*+ RULE */ или /*+ **ALL_ROWS** */). запрос выполнялся бы намного быстрее. Стабилизация плана оптимизатора позволяет это сделать. Можно независимо создавать и сохранять оптимальные шаблоны запросов **независимо** от существующего приложения. Затем с помощью триггера базы данных **ON LOGON** (который позволяет выполнить фрагмент кода при регистрации пользователя на сервере) или аналогичного механизма заставить существующее приложение использовать хранимый шаблон запроса.

Предположим, с помощью **SQL_TRACE** был получен SQL-оператор, выполняемый в приложении или при генерации отчета. Затем с помощью утилиты **TKPROF** был проанализирован соответствующий трассировочный файл, и оказалось, что запрос выполняется крайне медленно. Изучив руководство *Designing and Tuningfor Performance* и поэкспериментировав с описанными в нем подсказками, удалось выяснить, что при установке режима оптимизации **FIRST_ROWS** запрос работает отлично, но если задать этот режим для всего приложения, общая производительность резко снижается. Итак, хотелось бы в режиме **FIRST_ROWS** оптимизировать этот единственный запрос, а остальные выполнять в стандартном режиме **CHOOSE.** Обычно достаточно добавить подсказку /*+ **FIRST_ROWS** */ для этого запроса. Но мы не можем этого сделать — запрос изменить нельзя. Можно, однако, выполнить оператор CREATE OUTLINE, как было показано ранее, чтобы создать шаблон с указанным именем, а затем поместить его в стандартный **(DEFAULT)** набор шаблонов или в определенную категорию шаблонов. Затем среда изменяется так, чтобы применялся сгенерированный таким образом план. Например, в данном случае можно было бы выполнить оператор, а затем создать шаблон плана запроса. После этого с помощью триггера ON LOGON можно включать использование этого хранимого шаблона при каждой регистрации пользователя приложения.

Создание шаблона может вызвать определенные сложности, поскольку текст запроса, для которого надо сгенерировать хранимый шаблон, должен с точностью до байта совпадать с текстом оператора в приложении. Ниже мы шаг за шагом продемонстрируем, как проще всего создать шаблон. Мы будем и дальше использовать запрос к таблице **EMP** — именно его необходимо выполнять в режиме оптимизации **FIRST_ROWS.** Остальные операторы приложения должны выполняться в режиме оптимизации CHOOSE. Допустим, имеется "приложение" со следующим кодом:

```
tkyte@TKYTE816> create or replace procedure show_emps
  2  as
  3  begin
  4        for x in (select ename, empno
  5                        from emp
  6                       where empno > 0)
  7        loop
  8       dbms_output.put_line(x.enpno       ||  ',' || x.ename);
  9        end loop;
 10  end;
 11  /

Procedure created.
```

Теперь выполним эту процедуру при установленном режиме **SQL_TRACE** и по отчету утилиты **TKPROF** выясним, что для запроса используется нежелательный план (подробнее о режиме SQL_TRACE и утилите **TKPROF,** а также об их использовании в различных средах см. в главе 10). В данном случае мы просто используем оператор **ALTER** SESSION, поскольку речь идет о PL/SQL-процедуре, которую можно выполнить в среде SQL*Plus:

```
tkyte@TKYTE816> alter session set sql_trace=true;

Session altered.

tkyte@TKYTE816> exec show_emps
7876,ADAMS

7521,WARD

PL/SQL procedure successfully completed.
```

Затем применим утилиту **TKPROF** к полученному файлу трассировки:

```
SELECT ENAME,EMPNO
FROM
 EMP   WHERE EMPNO > 0

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        2      0.01       0.01          0          0          0           0
Execute      2      0.00       0.00          0          0          0           0
Fetch       30      0.00       0.00          0         36         24          28
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       34      0.01       0.01          0         36         24          28

Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 224      (recursive depth: 1)

Rows     Row Source Operation

      14  TABLE ACCESS FULL EMP
```

Прежде всего, обратите внимание, что формат запроса в результатах **TKPROF** совершенно другой, чем в приложении. Это — побочный эффект того, как в языке PL/SQL обрабатываются SQL-операторы: все статические SQL-операторы переписываются, и полученный запрос может выглядеть совсем не так, как запрос в исходном коде. При создании хранимого шаблона необходимо убедиться, что используется именно тот запрос, который поступил в базу данных, поскольку при стабилизации плана оптимизатора выполняется буквальное сравнение строк — должен использоваться **такой же** запрос, как в приложении, вплоть до пробелов, символов табуляции и новой строки. Однако ни текст в PL/SQL-процедуре, ни текст в отчете **TKPROF** не подходит! К счастью, можно использовать сами механизмы создания хранимых шаблонов для перехвата запроса, с которым надо работать. Мы включим неявную генерацию шаблонов, при этом текст SQL-запросов, поступающих в базу данных, будет перехвачен автоматически:

```
tkyte@TKYTE816> alter session set create_stored_outlines = hr_application;

Session altered.

tkyte@TKYTE816> exec  show_emps
7876,ADAMS


7521,WARD

PL/SQL procedure successfully completed.

tkyte@TKYTE816> alter session set create_stored_outlines = FALSE;

Session altered.

tkyte@TKYTE816> set long 50000
tkyte@TKYTE816> select name,  sql_text
  2     from user_outlines
  3    where category =  'HR_APPLICATION'
  4  /

NAME                           SQL_TEXT

SYS_OUTLINE_0104120951400008  SELECT ENAME,EMPNO   FROM EMP   WHERE EMPNO > 0
```

Чтобы включить автоматическую генерацию хранимых шаблонов для категории **HR_APPLICATION**, мы использовали оператор **ALTER SESSION**. После этого запустили приложение.

Команда **SET LONG** использована для того, чтобы в среде **SQL*Plus** был показан весь SQL-запрос; по умолчанию будет выдаваться только первых 80 байт.

Для получения тех же результатов можно использовать триггер базы данных **ON LOGON**, например такой:

```
tkyte@TKYTE816> create or replace trigger tkyte_logon
  2  after logon on database
  3  begin
  4      if  (user =  'TKYTE')  then
  5          execute  immediate
  6                  'alter  session  set use_stored_outlines  =
                                  hr_application';
  7     end if;
  8  end;
  9  /

Trigger created.
```

Для создания триггера на событие LOGON необходимы привилегии CREATE TRIGGER и ADMINISTER DATABASE TRIGGER. Кроме того, владельцу триггера необходимо наличие привилегии ALTER SESSION, предоставленной непосредственно, а не через роль.

Этот подход следует использовать для приложений, в которых нельзя выполнить оператор ALTER SESSION никаким другим способом.

Итак, теперь текст запроса имеется, и все готово для генерации хранимого шаблона с планом, который мы хотим использовать для запроса. Интересно отметить, что текст отличается от использовавшегося в PL/SQL-коде: все символы переведены в верхний регистр. Он отличается и от теста запроса в отчете **TKPROF:** там использовались символы новой строки. Поскольку использование хранимого шаблона зависит от **точного** совпадения запросов, я собираюсь продемонстрировать, как проще всего изменить набор подсказок, связанных с перехваченным шаблоном. Обратите внимание, как в представленных выше результатах запроса к представлению **USER_OUTLINES** мы выбрали столбцы **NAME** и **SQL_TEXT,** чтобы можно было определить интересующий нас запрос и найти имя соответствующего хранимого шаблона, **SYS_OUTLINE_0104120951400008.**

Итак, можно задать цель оптимизации **FIRST_ROWS,** пересоздать шаблон с соответствующим именем, и все:

```
tkyte@TKYTE816> alter session set optimizer_goal=first_rows
  2  /

Session altered.

tkyte@TKYTE816> alter outline SYS_OUTLINE_0104120951400008 rebuild
  2  /

Outline altered.

tkyte@TKYTE816> alter session set optimizer_goal=choose;

Session altered.
```

Мы начали с установки режима оптимизации **FIRST_ROWS,** вместо **CHOOSE.** Известно, что если выполнять запрос в режиме оптимизации **FIRST_ROWS,** будет выбран необходимый план (именно такой сценарий мы подготовили для демонстрации, в реальной же ситуации это надо определить с помощью нескольких итераций тестирования и настройки). Вместо повторного выполнения запроса мы просто перестроим **(REBUILD)** шаблон — при перестройке будет сгенерирован план, соответствующий текущей среде.

Теперь, чтобы убедиться, что шаблоны работают, надо включить использование соответствующей категории шаблонов. В целях демонстрации мы будем использовать оператор **ALTER SESSION** в интерактивном сеансе, но чтобы делать это автоматически и без изменения кода приложения, можно использовать триггер **ON LOGON** для включения поддержки шаблонов при регистрации пользователя. Чтобы проверить, используется ли шаблон, придется повторно запустить приложение:

```
tkyte@TKYTE816> alter session set optimizer_goal=choose;

Session altered.

tkyte@TKYTE816> alter session set USE_STORED_OUTLINES = hr_application;

Session altered.

tkyte@TKYTE816> alter session set sql_trace=true;

Session altered.
```

```
tkyte@TKYTE816> exec show_emps
7369,SMITH


7934,MILLER

PL/SQL procedure successfully completed.
```

Мы восстановили стандартный режим оптимизации, заставили сервер Oracle использовать хранимые шаблоны из категории HR_APPLICATION и еще раз выполнили приложение. Теперь в отчете утилиты TKPROF можно обнаружить следующее:

```
SELECT ENAME,EMPNO
FROM
  EMP   WHERE  EMPNO  >  0


call       count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
Parse          1      0.01        0.01           0           0           1           0
Execute        1      0.00        0.00           0           0           0           0
Fetch         15      0.00        0.00           0          28           0          14
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
total         17      0.01        0.01           0          28           1          14

Misses In library cache during parse:  1
Optimizer goal: CHOOSE
Parsing user id:  224      (recursive depth:  1)

Rows      Row Source Operation

      14  TABLE ACCESS BY INDEX ROWID EMP
      15    INDEX RANGE SCAN  (object id 28094)
```

Это доказывает, что использовался нужный план. При выполнении приложения оператор ALTER SESSION SET USE_STORED_OUTLINE подключил соответствующие хранимые шаблоны. Интересующий нас запрос будет оптимизирован с использованием запомненных подсказок, сгенерированных в режиме оптимизации FIRST_ROWS. Остальные запросы приложения будут оптимизироваться так же, как раньше.

Средство разработки

Предположим, создается новое приложение, которое необходимо передать большому количеству пользователей. Контролировать среду базы данных у пользователей практически невозможно — приложение может оказаться единственным или одним из десятка других работающих приложений. У используемых серверов могут быть разные значения параметров инициализации, влияющих на работу оптимизатора, например DB_BLOCK_BUFFERS, DB_FILE_MULTIBLOCK_READ_COUNT, HASH_MULTIBLOCK_IO_COUNT, OPTIMIZER_GOAL, HASH_JOIN_ENABLED и т.п. Может быть включена или отключена возможность параллельного выполнения запросов. Сервер на большой машине может иметь как большую, так и маленькую область SGA. Приложение может работать на сервере версии 8.1.6.1, 8.1.7 или 8.1.6.2. И так да-

лее. На генерируемый оптимизатором план выполнения запроса может влиять множество факторов.

При разработке приложения много усилий было затрачено на то, чтобы оно обращалось к данным "правильно". На тестовых примерах и при проверке масштабируемости на реальных данных большого объема, на машинах разработчиков, приложение работает замечательно. Все происходит именно так, как предполагалось. Так почему же пользователи звонят в службу поддержки и жалуются на низкую производительность? Дело в том, что на их машинах и для их конфигураций оптимизатор использует другие планы выполнения запросов.

Чтобы справиться с этой проблемой, можно использовать стабилизацию плана оптимизатора. Настроив приложение в среде разработки и тщательно протестировав его на реальных данных (с соответствующим количеством и распределением строк в таблицах), вы должны сделать последний шаг — сгенерировать шаблоны для всех запросов. Это легко сделать с помощью триггера ON LOGON, на этот раз со следующим кодом:

```
sys@TKYTE816> create or replace trigger tkyte_logon
  2  after logon on database
  3  begin
  4      if (user = 'TKYTE') then
  5          execute immediate
  6                  'alter session set create_stored_outlines = KillerApp';
  7      end if;
  8  end;
  9  /

Trigger created.
```

Триггер был создан от имени пользователя SYS, который стандартно имеет все необходимые для создания такого триггера привилегии.

Теперь для каждого выполняемого запроса будет автоматически создаваться шаблон. Он будет получать соответствующее имя и сохраняться в категории **KillerApp.** Необходимо выполнить полный набор тестов для приложения, чтобы были задействованы **все** SQL-операторы. В результате этого будут собраны шаблоны для всех запросов в приложении. После этого с помощью утилиты EXP можно экспортировать шаблоны и установить как часть процедуры импорта данных с помощью утилиты **IMP** (эта процедура будет описана далее в этой главе, в разделе "Перенос шаблонов из одной базы данных в другую"). Приложение сразу после подключения к базе данных должно выполнять следующую команду:

```
alter session set use_stored_outlines = KillerApp;
```

Так можно гарантировать, что оптимальные планы запросов, над которыми вы так долго работали, используются независимо от установок на машинах клиентов. Это пригодится не только для "внешних" клиентов, но и при переносе приложения с тестового сервера на производственный. При этом сократится количество жалоб: "Все прекрасно работает на тестовом сервере, но при переносе на производственный работа резко замедляется". После этого обычно добавляют: "А ведь серверы — абсолютно одинаковые".

И только потом оказывается, что у них — разный объем оперативной памяти, разное количество процессоров и параметры в файле init.ora имеют различные значения, в соответствии с разными конфигурациями оборудования. Любой из этих факторов может влиять на оптимизатор, и приводит к изменению плана. Стабилизация плана оптимизатора позволяет избежать этой проблемы.

Следует отметить, что при этом можно упустить преимущества, предоставляемые последними, более совершенными версиями оптимизатора. При установке программного обеспечения с новыми возможностями, имеет смысл отключить использование шаблонов при разработке и тестировании приложения, чтобы выявить запросы, которые могут работать быстрее по новому плану, предложенному оптимизатором.

Проверка использования индексов

На самом деле хранимые шаблоны создавались с другой целью; это скорее побочный эффект, но их можно так использовать! Часто задают вопрос: "В базе данных — множество индексов, и некоторые из них, определенно, не используются, но непонятно — какие. Как это определить?". Один из способов — с помощью хранимых шаблонов: в них перечислены имена всех индексов, использованных в плане выполнения запроса. Если используется триггер ON LOGON для включения автоматической генерации шаблонов, поработайте некоторое время с системой, а затем отключите генерацию, — будет получен сравнительно полный список индексов, используемых в системе (и запросов, которые эти индексы используют). Как будет показано ниже, все "подсказки", используемые хранимыми шаблонами, находятся в таблице словаря данных. Благодаря этому легко понять, какие индексы используются (и какими запросами), а какие — нет. Например, по результатам предыдущих двух примеров можно определить, какие запросы используют индекс EMP_PK:

```
tkyte@TKYTE816> select name, hint
  2   from user_outline_hints
  3   where hint like  'INDEX%EMP_PK%'
  4   /

KAME               HINT
-----------------  ----------------------
MYOUTLINE          INDEX(EMP EMP_PK)
FIRST ROWS EMP     INDEX(EMP EMP PK)
```

Столбец NAME этого запроса позволяет найти в представлении **USER_OUTLINES** текст исходного SQL-запроса, использующего индекс.

Получение списка SQL-операторов, выполненных приложением

Это тоже побочный эффект, а не прямое назначение хранимых шаблонов, но тем не менее его можно использовать. Часто пользователи интересуются, какие **SQL**-операторы фактически выполняют их приложения. Изменить текст приложения нельзя, а установка **SQL_TRACE ON** приводит к слишком большим расходам ресурсов. Используя

триггер **ON LOGON** для некоторых пользователей приложения, можно автоматически сохранить в таблицах **OUTLINE** все SQL-операторы, выполненные приложением. Эту информацию в дальнейшем можно использовать для настройки или анализа.

Следует помнить, что при этом SQL-операторы будут сохраняться по мере выполнения. Чтобы получить исчерпывающий список SQL-операторов, которые может выполнить приложение, надо заставить приложение все эти операторы выполнить, т.е. применить все средства и выполнить функции приложения во всех возможных сочетаниях.

Как выполняется стабилизация плана оптимизатора

Стабилизация плана оптимизатора выполняется на базе механизма "подсказок" Oracle. Используя предыдущий пример с таблицей EMP, мы сможем увидеть подсказки, сохраненные для запроса, а также их применение во время выполнения. Мы также рассмотрим пользовательскую схему **OUTLN,** в которой хранятся все шаблоны запросов и подсказки для них.

Первый шаг использования стабилизации планов оптимизатора — получение шаблона запроса. Поскольку ранее мы уже делали это с помощью оператора **CREATE OUTLINE,** перейдем к тому, как сервер обрабатывает эти шаблоны.

Представления OUTLINES и OUTLINE_HINTS

С шаблонами запросов связаны два представления, между которыми есть отношение главное/подчиненное. Главное представление — **OUTLINES** (как обычно, есть три его версии: **DBA_, ALL_** и **USER_.** Подчиненное представление — **OUTLINE_HINTS** (оно тоже доступно в трех версиях). В следующих разделах описаны эти представления и их использование.

Представления _OUTLINES

В этих представлениях находятся хранимые шаблоны. В представлении **DBA_OUTLINES** есть записи для всех хранимых шаблонов в системе, тогда как в представлениях **ALL_** и **USER_OUTLINES** присутствуют только строки, имеющие отношение к текущему пользователю (шаблоны, **доступные** или **созданные** пользователем, соответственно). Поскольку представления **DBA_OUTLINES** и **USER_OUTLINES** отличаются только одним столбцом (в представлении **DBA** есть столбец **OWNER,** содержащий имя схемы, в которой создан шаблон), мы рассмотрим представление **DBA_OUTLINES:**

- **NAME.** Имя шаблона, заданное в операторе **CREATE** OUTLINE (в представленных выше примерах использовались имена **MYOUTLINE** и **FIRST_ROWS_EMP).** Если для создания хранимого шаблона использовался оператор **ALTER SESSION** (этот метод мы детально рассмотрим далее в этой главе), имя шаблону система генерирует автоматически. Следует заметить, что имя шаблона — уникально (имя шаблона является первичным ключом). Нельзя создать шаблон с одним и тем же

именем в двух категориях или у различных пользователей. Более детально это описано в разделе "Проблемы".

- **OWNER.** Схема, в которой создан шаблон. Шаблоны не "принадлежат" никому, так что имя столбца — несколько неправильное. Правильно было бы назвать столбец **CREATOR,** создатель.
- **CATEGORY.** Категория, к которой отнесена схема (в нашем примере — **MYCATEGORY).** Шаблоны запросов могут принадлежать к категории, указанной по имени, либо к общей категории **DEFAULT,** которая используется, если имя категории не задано. В ходе работы пользователь или приложение выполняет оператор **ALTER SESSION SET USE_STORED_OUTLINES = <TRUE|имя_категории>,** чтобы указать, какой набор хранимых шаблонов надо использовать. При установке значения **TRUE** будут использоваться шаблоны стандартной категории, **DEFAULT. В** каждый момент времени может использоваться только одна категория шаблонов.
- **USED.** Этот атрибут показывает, был ли указанный шаблон хоть раз использован. Он будет иметь значение **unused** до первого использования шаблона для изменения плана выполнения запроса; при этом атрибут получает значение **used.**
- **TIMESTAMP.** Дата и время создания исходного шаблона.
- **VERSION.** Версия СУБД, в которой был создан исходный шаблон.
- **SQL_TEXT.** Фактический (дословный) SQL-запрос, использованный для генерации шаблона. Этот шаблон может использоваться только для запросов, текст которых полностью совпадает.

Итак, например, после выполнения запросов в представленных выше примерах, в представлении **USER_OUTLINES** будет следующая информация:

```
tkyte@TKYTE816> select * from user_outlines;

KAME         CATEGORY          USED   TIMESTAMP  VERSION    SQLTEXT

MYOUTLINE    MYCATEGORY        USED   11-APR-01  8.1.6.0.0  select empno, ename
                                                             from emp where empno
                                                             > 0

FIRST_ROWS_EMP HR_APPLICATION  USED   12-APR-01   8.1.6.0.0 SELECT ENAME,EMPNO
                                                             FROM EMP WHERE EMPNO
                                                             > 0
```

Как и ожидалось, выданы все описанные выше атрибуты.

Представления _OUTLINE_HINTS

В этих представлениях находятся реальные подсказки, которые надо применять на разных внутренних стадиях плана выполнения запроса. Сервер по ходу работы переписывает переданный запрос, встраивая эти подсказки в соответствующие места, что и дает необходимый план выполнения. В тексте запроса эти подсказки не появляются, — они добавляются во внутренние структуры плана выполнения запроса. Един-

ственное структурное отличие между представлением **DBA_OUTLINE_HINTS**, **USER_OUTLINE_HINTS** и **ALL_OUTLINE_HINTS** - добавление столбца **OWNER**, идентифицирующего пользователя, создавшего шаблон.

- **NAME**. Имя хранимого шаблона. Если шаблон создан с помощью оператора **CREATE OUTLINE**, это будет имя, заданное в операторе. Если шаблон сгенерирован автоматически с помощью **ALTER SESSION**, идентификатор присваивается системой, и будет иметь вид **SYS_OUTLINE_0104120957410010**, как у шаблона в нашем примере.
- **OWNER**. Имя пользователя, создавшего шаблон запроса.
- **NODE**. Запрос или подзапрос, к которому применяется подсказка. Запрос верхнего уровня получает значение 1 в столбце NODE, а последующие подзапросы, встроенные в основной запрос, получают последовательно увеличивающиеся значения.
- **STAGE**. Стадия выполнения, на которой применяются подсказки в ходе обработки запроса. Это число представляет стадию, на которой подсказка будет "вписана" в запрос. Речь идет о внутренних стадиях обработки, выполняемых оптимизатором Oracle, которые обычно пользователям недоступны.
- **JOIN_POS**. Задает таблицу, к которой будет применяться эта подсказка. Для всех подсказок, не задающих метод доступа, в этом столбце будет значение ноль. Для подсказок, задающих метод доступа (например, доступ к таблице по индексу), столбец **JOIN_POS** задает таблицу.
- **HINT**. Подсказка, которая должна быть встроена в запрос.

Посмотрим на результаты выполнения исходного примера:

```
tkyte@TKYTE816> break on stage skip 1

tkyte@TKYTE816> select stage, name, node, join_pos, bint
  2      from user_outline_hints
  3    where name = 'MYOUTLINE'
  4    order by stage
  5  /

 STAGE NAME          NODE     JOIN_POS HINT
------------------------------------------------------------
       MYOUTLINE        1            0 NOREWRITE
       MYOUTLINE        1            0 RULE
     2 MYOUTLINE        1            0 NOREWRITE
     3 MYOUTLINE        1            0 NO_EXPAND
       MYOUTLINE        1            0 ORDERED
       MYOUTLINE        1            0 NO_FACT(EMP)
       MYOUTLINE        1            1 INDEX(BMP EMP_PK)

7 rows selected.
```

Запрос показывает, что на стадии 1 сервер применяет подсказки NOREWRITE и RULE. Подсказка NOREWRITE предотвращает перезапись запроса по ходу выполнения (если кто-то в дальнейшем добавит соответствующие средства или включит пара-

метр сеанса/системы, обуславливающий вызов метода **QUERY_REWRITE**). Подсказка **RULE** требует использовать оптимизатор, основанный на правилах, независимо от текущего значения **OPTIMIZER_GOAL** и наличия (или отсутствия) статистической информации о таблице.

На стадии 2 снова предотвращается перезапись запроса. На стадии 3 вставляются подсказки, действительно определяющие требуемый план выполнения запроса. Применяется подсказка **ORDERED**, которая требует учитывать порядок следования таблиц в конструкции **FROM** при соединении (поскольку в нашем примере используется одна таблица, эта подсказка излишня). Далее применяется подсказка **NO_EXPAND** (она применяется для условий, связанных с объектами, а, поскольку объекты в запросе не задействованы, эта подсказка не нужна). Затем, применяется внутренняя, не описанная подсказка **NO_FACT**. Наконец, к таблице 1 (столбец **JOIN_POS**) применяется подсказка **INDEX()**, задающая метод доступа, — для доступа к таблице **EMP** используется индекс **EMP_PK**, первичный ключ.

Итак, вот как выполняется стабилизация плана оптимизатора. План сохраняется в указанной или стандартной категории. При выполнении приложение выбирает "использование" определенной категории планов, и оптимизатор добавляет соответствующие подсказки в текст запроса, чтобы получаемый план каждый раз был одинаковым.

Создание хранимых шаблонов

Есть два способа генерации планов. Мы уже бегло представили их в предыдущих примерах. Один из способов подразумевает использование оператора ЯОД, а второй — установку параметра сеанса. Мы рассмотрим оба способа и опишем, когда имеет смысл использовать каждый из них. В любом случае, однако, надо убедиться, что пользователь, создающий шаблоны, имеет соответствующие привилегии для создания и управления шаблонами.

Привилегии, необходимые для создания хранимых шаблонов

Создавать и использовать хранимые шаблоны могут пользователи, обладающие следующими четырьмя привилегиями.

- **CREATE ANY OUTLINE**. Позволяет создавать шаблоны в базе данных. При отсутствии этой привилегии будет выдаваться сообщение об ошибке **ORA-18005: create any outline privilege is required for this operation**.
- **ALTER ANY OUTLINE**. Позволяет изменять (переименовывать, изменять категорию или пересоздавать план) шаблон запроса.
- **DROP ANY OUTLINE**. Позволяет удалять существующий шаблон с указанным именем.
- **EXECUTE ON OUTLN_PKG**. Позволяет выполнять подпрограммы пакета **OUTLINE** (подробнее о его возможностях см. далее).

Обратите внимание, что это привилегии класса **ANY.** Это означает, что при наличии привилегии **CREATE** OR **REPLACE ANY OUTLINE** можно переписать шаблон другого пользователя, не спрашивая у него разрешения. Шаблоны, в отличие от большинства других объектов базы данных не принадлежат никому. У шаблона есть создатель, но нет владельца в обычном смысле. Если можно удалять собственные шаблоны, то можно (ненамеренно) удалить и шаблон любого другого пользователя, поэтому при использовании этих привилегий надо быть внимательным. Подробнее об этом см. в подразделе "Пространство имен для шаблонов — глобальное" раздела "Проблемы".

Использование операторов ЯОД

Для создания хранимых шаблонов можно использовать оператор ЯОД следующей структуры:

```
CREATE [OR REPLACE] OUTLINE имя_шаблона
[FOR CATEGORY имя_категории]
ON оператор_для_которого_сохраняется_шаблон
```

В этом операторе:

- **Имя_шаблона** — имя, присвоенное шаблону. Оно должно иметь смысл для создателя и разработчика приложения. При этом на имя налагаются те же ограничения, что и для любого объекта базы данных (не более 30 символов, начинается с буквы и т.д.). Кроме того, **имя_шаблона** должно быть уникальным для базы данных, а не для пользователя или категории, как можно было бы подумать, поэтому будьте особенно внимательны при использовании конструкции **OR REPLACE,** поскольку оператор перезапишет любой существующий шаблон с таким именем.
- **Имя_категории** — имя, используемое для группировки шаблонов. Эта часть оператора **CREATE** — не обязательная, и если категория не задана, шаблон будет отнесен к категории **DEFAULT.** Рекомендуется явно указывать имя категории и не использовать категорию **DEFAULT.** Поскольку сеанс в каждый момент времени может использовать только одну категорию шаблонов, в ней надо сохранить шаблоны планов для всех существенных запросов.
- **Оператор_для_которого_сохраняется_шаблон** — любой допустимый SQL-оператор ЯМД.

Генерация шаблонов с помощью операторов ЯОД больше всего подходит для приложений, в которых все SQL-операторы хранятся вне приложения. Другими словами, есть файл ресурсов, в котором записаны все потенциально выполняемые SQL-операторы. В этом случае по такому файлу очень легко сгенерировать сценарий с операторами **CREATE OUTLINE** и выполнить его на сервере. Это гарантирует создание шаблонов для всех запросов (если запросы указаны в этом файле ресурсов). Кроме того, такой подход предохраняет от случайной генерации шаблонов для лишних запросов. Например, если используется триггер **ON LOGON,** после регистрации в SQL*Plus окажется, что для автоматически выполняемых утилитой SQL*Plus запросов тоже сохранены шаблоны.

Кроме того, операторы ЯОД используются, если надо сгенерировать шаблоны только для небольшого количества запросов. Например, этот подход пригоден при использовании шаблонов как средства настройки. Итак, если надо сгенерировать хранимые шаблоны только для небольшой части запросов приложения, это имеет смысл делать с помощью операторов ЯОД.

Использование оператора ALTER SESSION

Это более универсальный метод генерации шаблонов запросов. Он применяется аналогично установке **SQL_TRACE** при трассировке программ. С момента выполнения соответствующего оператора **ALTER SESSION** и до отключения создания хранимых шаблонов для каждого выполняемого запроса будет сохраняться шаблон (почти для каждого — в разделе "Проблемы" описаны случаи, когда шаблон не сохраняется).

Этот метод можно применять для любого приложения, если требуется стабилизировать все планы. Другими словами, когда необходимо точно знать, какими будут планы выполнения SQL-операторов, независимо от версии сервера, на котором будет установлено приложение, независимо от значений параметров инициализации экземпляра и т.п. Чтобы добиться этого для приложения, можно использовать триггер **ON LOGON**, как было показано ранее, а затем полностью протестировать приложение, выполнив все возможные запросы. Это надо сделать на тестовом сервере в процессе окончательного тестирования перед поставкой приложения клиентам. После сбора всех планов необходимо извлечь их с помощью утилиты EXP, а затем устанавливать с помощью утилиты IMP в процессе инсталляции приложения. Этот процесс подробно описан далее в разделе "Перенос шаблонов из одной базы данных в другую".

Этот метод также используется, если на уровне сервера включена *автоматическая подстановка связываемых переменных* (auto binding). В разделе "Проблемы" взаимодействие автоматической подстановки связываемых переменных со стабилизацией плана оптимизатора описано более подробно.

Синтаксис соответствующих версий оператора **ALTER SESSION** несложен:

```
ALTER SESSION SET CREATE_STORED_OUTLINES = TRUE;
ALTER SESSION SET CREATE_STORED_OUTLINES = FALSE;
ALTER SESSION SET CREATE_STORED_OUTLINES = категория_шаблонов;
```

Если параметр **CREATE_STORED_OUTLINES** получает значение **TRUE**, сервер Oracle будет генерировать хранимые шаблоны для категории **DEFAULT**. Категория **DEFAULT** — самая обычная категория с соответствующим именем; ее использование надо включать так же, как и любой другой категории. После установки параметру **CREATE_STORED_OUTLINES** значения **FALSE** сервер Oracle перестанет генерировать хранимые шаблоны для соответствующего сеанса.

Если параметру **CREATE_STORED_OUTLINES** задано значение **категория_шаблонов,** сервер Oracle будет генерировать шаблоны для всех выполняемых запросов и сохранять их в указанной категории. Именно так предпочтительнее использовать этот метод. Для ясности рекомендуется, чтобы каждое приложение использовало собственную категорию шаблонов, особенно если предполагается его установка в базе данных, с которой работает множество других приложений, тоже использующих стабилизацию плана оптимизатора. Это предотвратит конфликты между приложениями и упростит поиск приложения, которому принадлежит шаблон.

Пользователь OUTLN

Пользовательская схема **OUTLN** теперь создается во всех базах данных Oracle8i, со стандартным паролем **OUTLN.** Администратор базы данных должен изменить пароль этого пользователя сразу после установки так же, как и для пользователей **SYS** и SYSTEM.

В этой схеме находятся две таблицы и ряд индексов, которые стандартно создаются в табличном пространстве **SYSTEM.** Если планируется интенсивно использовать стабилизацию плана запроса (особенно, если для запоминания планов запросов предполагается использовать метод **ALTER SESSION),** имеет смысл перенести эти объекты из табличного пространства **SYSTEM** в другое, специально созданное табличное пространство. Одна из этих двух таблиц содержит столбец типа **LONG,** поэтому перенести ее с помощью оператора **ALTER TABLE имя_таблицы MOVE** не удастся. Вместо этого придется экспортировать и импортировать эти объекты. Ниже описана последовательность действий для переноса всей схемы **OUTLN** из табличного пространства **SYSTEM** в табличное пространство **TOOLS.**

1. Экспортирование схемы пользователя **OUTLN:**

```
exp userid=outln/outln owner=outln
```

2. Изменение стандартного табличного пространства пользователя **OUTLN** с **SYSTEM** на **TOOLS** и установка ему неограниченной квоты на табличное пространство **TOOLS** и квоты Ok на табличное пространство **SYSTEM:**

```
alter user outln default tablespace tools;
revoke unlimited tablespace from outln;
alter user outln quota Ok on system;
alter user outln quota unlimited on tools;
```

3. Удаление таблиц **OL$** и **OL$HINTS** из пользовательской схемы **OUTLN:**

```
drop table ol$;
drop table ol$hints;
```

4. Импортирование пользовательской схемы **OUTLN:**

```
imp userid=outln/outln full=yes
```

Учтите, что если в системе уже использовались шаблоны, описанные выше действия должны выполняться в однопользовательском режиме или в базе данных, с которой пользователи не работают активно.

Имеет смысл периодически контролировать использование пространства для таблиц **OL$** и **OL$HINTS,** а также для соответствующих индексов.

Перенос шаблонов из одной базы данных в другую

Разобравшись, как перенести всю схему **OUTLN** из одного табличного пространства в другое, давайте посмотрим, как экспортировать шаблоны запросов из базы данных разработчиков и импортировать в другую базу данных. Это приходится делать, если шаблоны используются для стабилизации планов оптимизатора в приложении, которое бу-

дет устанавливаться у клиентов, или при переносе приложения с тестового сервера в производственную среду.

Проще всего сделать это с помощью файла параметров экспорта (чтобы избежать проблем с маскировкой специальных символов в командном интерпретаторе и использованием командной строки в NT). Я создал файл параметров для экспорта **exp.par** со следующим содержимым:

```
query="where  category='HR_APPLICATION'"
tables=(ol$,ol$hints)
```

Это приведет к экспортированию всех хранимых шаблонов из категории **HR_APPLICATION.** Изменять в файле **exp.par** придется только имя переносимой категории. После этого надо выполнить:

```
exp userid=outln/<пароль> parfile=exp.par
```

заменив <пароль> соответствующим паролем пользователя **OUTLN.** При этом будут экспортированы только интересующие нас строки. Полученный файл **expdat.dmp** надо перенести на целевой сервер (или выполнить IMP по сети) и импортировать полностью. Для этого используется команда:

```
imp  userid=outln/outln  full=y  ignore=yes
```

В этом случае надо использовать конструкцию **IGNORE=YES,** поскольку мы добавляем строки в существующую таблицу, а не просто переносим таблицу, как в предыдущем разделе. Средства стабилизации плана оптимизатора поддерживают экспорт и импорт, так что можно это делать. Не надо изменять таблицы **OL$** и **OL$HINT** непосредственно, но можно переносить их из одной базы данных в другую с помощью утилит экспорта/импорта. Фактически **EXP/IMP** — единственное средство для безопасного решения этой задачи. Рассмотрим, что произойдет, если хранимый шаблон **MYOUTLINE** был экспортирован и перенесен в другую базу данных, где **уже** есть шаблон **MYOUTLINE.** Если для этого использовалось другое средство, а не утилиты EXP/IMP, шаблон в базе данных будет испорчен. Предположим, данные копируются с помощью SQL-операторов. Некоторые данные будут скопированы, другие — нет из-за конфликтов по первичному ключу. В конечном итоге в базе данных окажется два набора хранимых шаблонов. Только утилиты экспорта и импорта позволяют правильно решить эту задачу и содержат специальные фрагменты кода, гарантирующие, что шаблон из одной базы данных не перезапишет шаблон в другой (подробнее это описано далее в разделе "Пакет OUTLN_PKG").

Получение нужного шаблона

Если средства стабилизации плана оптимизатора используются для настройки, возникает вопрос: как получить от оптимизатора нужный план? Ранее я демонстрировал, как, изменяя параметры сеанса и выполняя конкретный запрос, можно сгенерировать "хороший" план. Это, несомненно, самый простой метод. Если возможно, надо просто установить соответствующие параметры сеанса с помощью оператора **ALTER SESSION,** а затем выполнить оператор **CREATE OUTLINE** для запроса. Если запрос уже упоминается в таблицах шаблонов, надо просто пересоздать для него план, как это было еде-

лано в одном из предыдущих примеров. Второй метод предпочтительнее, поскольку гарантирует совпадение текста запроса. Первый тоже можно использовать во многих случаях, в частности:

- Необходимо, чтобы определенный запрос использовал конкретный режим оптимизации, независимо от значения параметра **OPTIMIZER_GOAL** в ходе выполнения приложения. Можно задать соответствующий режим оптимизации с помощью оператора **ALTER SESSION** и выполнить оператор **CREATE OUTLINE** для соответствующего запроса.
- Необходимо избежать использования определенных возможностей, например **QUERY_REWRITE_ENABLED**, **HASH_JOIN_ENABLED**, или перехода к прежним значениям **OPTIMIZER_FEATURES_ENABLED.** Можно создать сеанс, выполнить ряд операторов **ALTER SESSION** для включения/отключения соответствующих возможностей, а затем выполнить операторы **CREATE OUTLINE** для требуемых запросов.

Хорошо, а что делать, если установки параметров на уровне сеанса, например **HASH_JOIN_ENABLED,** не изменяют план выполнения запросов нужным образом? Когда единственный способ, дающий необходимый результат, — физическое включение подсказки в текст запроса? Можно, конечно, использовать подсказки в запросах, план выполнения которых надо стабилизировать, но это не то, что требуется. Необходимо, чтобы этот план использовал запрос, выполняющийся без подсказок. Но для того чтобы была возможность применять этот хранимый план, должны выполняться запросы, **в точности** с тем же текстом, который использовался для генерации шаблона. Надо сохранить шаблон для запроса без подсказок и использовать соответствующий план при выполнении этого запроса в дальнейшем. Зная, как выполняется стабилизация плана оптимизатора, это можно сделать, хотя и несколько нетривиальным способом. Вот как это делается.

Предположим, надо сохранить шаблон для следующего запроса:

```
scott@TKYTE816> set autotrace traceonly explain
scott@TKYTE816> select * from emp, dept where emp.deptno = dept.deptno;

Execution Plan

    0      SELECT STATEMENT Optimizer=CHOOSE
    1    0   NESTED LOOPS
    2    1     TABLE ACCESS (FULL) OF 'EMP'
    3    1     TABLE ACCESS (BY INDEX ROWID) OF 'DEPT'
    4    3       INDEX (UNIQUE SCAN) OF 'DEPT_PK' (UNIQUE)
```

При тестировании и настройке оказалось, что следующий запрос работает намного лучше:

```
scott@TKYTE816> select *
  ' 2     from (select /*+ use_hash(emp) */ * from emp) emp,
   3           (select /*+ usehash(dept) */ * from dept) dept
   4  where emp.deptno = dept.deptno
   5  /
```

```
Execution  Flan
----------------------------------------------------------
    0      SELECT STATEMENT Optimizer=CHOOSE  (Cost=3 Card=67 Bytes=7839)
    1    0   HASH JOIN (Cost=3 Card=67 Bytes=7839)
    2    1     TABLE ACCESS (FULL) OF 'EMP'  (Cost=l Card-82 Bytes=7134)
    3    1     TABLE ACCESS (FULL) OF 'DEPT' (Cost-1 Card=82 Bytes=2460)
```

Производительность несравнимо выше, чем при соединении вложенными циклами. *Хотелосьбы, чтобы приложения, использующие первый запрос, получали план с соединением хешированием, а не вложенным циклом, но без изменения кода приложений* (какая бы ни была причина, мы не можем добавлять подсказки в код).

Ну, поскольку использование шаблонов планов запроса основано на сравнении строк, можно добиться требуемого результата с помощью другой схемы и представлений с подсказками. Поскольку использованные выше объекты находятся в пользовательской схеме SCOTT, мы создадим представления в пользовательской схеме TKYTE:

```
scott@TKYTE816> grant select on emp to tkyte;

Grant succeeded.

scott@TKYTE816> grant select on dept to tkyte;

Grant succeeded.

scott@TKYTE816> connect tkyte/tkyte
Connected.

tkyte@TKYTE816> drop table emp;

Table dropped.

tkyte@TKYTE816> drop table dept;

Table dropped.

tkyte@TKYTE816> create or replace view emp as
  2  select /*+ use_hash(emp) */ * from acott.emp emp

  3  /

View created.

tkyte@TKYTE816> create or replace view dept as
  2  select /*+ use_hash(dept) */  * from scott.dept dept
  3  /
View created.
```

Теперь сгенерируем хранимый шаблон для запроса в приложении:

```
tkyte@TKYTE816> create or replace outline my_outline
  2  for category my_category
  3  on select * from emp, dept where emp.deptno = dept.deptno;

Outline created.
```

Итак, в пользовательской схеме TKYTE имеются *представления с подсказками* для базовых объектов, и мы создали в этой схеме хранимый шаблон для требуемого запроса. При желании теперь можно удалить представления. У нас уже есть все необходимое: хранимый шаблон, использующий *соединения хешированием*. Теперь, снова зарегистрировавшись как SCOTT, мы делаем следующее:

```
scott@TKYTE816> connect scott/tiger

scott@TKYTE816> alter session set use_stored_outlines=my_category;

Session altered.

scott@TKYTE816> set autotrace traceonly explain
scott@TKYTE816> select * from emp, dept where emp.deptno = dept.deptno;
Execution Plan

    0       SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=67 Bytes=7839)
    1    0    HASH JOIN (Cost=3 Card=67 Bytes=7839)
    2    1      TABLE ACCESS (FULL) OF 'EMP' (Cost=1 Card=82 Bytes=7134)
    3    1      TABLE ACCESS (FULL) OF 'DEPT' (Cost=1 Card=82 Bytes=2460)
```

Используя соответствующую категорию шаблонов, мы получили нужный план. Дело в том, что механизм стабилизации плана оптимизатора не разрешает ссылаться на объекты в тексте SQL-операторов. Просто запоминается строка запроса, и, если получается другая строка, в точности совпадающая с сохраненной в шаблоне указанной категории, оптимизатор использует сохраненные подсказки. Именно так и было задумано изначально.

Проверяя совпадение строк, можно применять представления и /или синонимы для создания шаблонов запросов, использующих при генерации окончательного плана нужные подсказки. С учетом возможностей метода, основанного на использовании оператора ALTER SESSION, можно утверждать, что сгенерировать можно большинство необходимых планов.

Управление шаблонами

Сейчас мы детально рассмотрим средства управления шаблонами: операторы ЯОД **(ALTER** и **DROP)** или подпрограммы стандартного пакета **OUTLN_PKG.**

Операторы ЯОД

Кроме оператора CREATE, для управления шаблонами запросов можно также использовать операторы **ALTER** и **DROP.** Оператор **ALTER** позволяет:

- переименовать **(RENAME)** хранимый шаблон;
- пересоздать **(REBUILD)** план для хранимого шаблона;
- изменить **(CHANGE)** категорию хранимого шаблона.

Оператор **DROP** удаляет указанный по имени хранимый шаблон.

Оператор ALTER OUTLINE

Оператор **ALTER** имеет три версии, и мы рассмотрим их поочередно. Чтобы разобраться, как работает этот оператор, создадим хранимый шаблон, а потом будем изменять его различными способами:

```
tkyte@TKYTE816> create or replace outline my_outline
  2   for category my_category
```

```
  3  on select * from all_objects
  4  /

Outline created.

tkyte@TKYTE816> select name, category, sql_text from user_outlines;

NAME                  CATEGORY        SQL_TEXT

MY_OUTLINE            M_CATEGORY      select * from
                                      all_objects

tkyte@TKYTE816> select count(*)  from user_outline_hints
                                        Where name = 'MY_OUTLINE' ;
  COUNT (*)

       138
```

Итак, мы работаем с шаблоном **MY_OUTLINE** в категории **MY_CATEGORY**, с которым сейчас связано 138 подсказок (у вас результат может быть другим, в зависимости от установок оптимизатора).

Прежде всего, оператор **ALTER OUTLINE** позволяет переименовать хранимый шаблон. Эта версия оператора имеет следующий синтаксис:

alter outline ***имя_шаблона*** **rename to новое_имя**

Итак, применим этот оператор для переименования шаблона с **MY_OUTLINE** в **PLAN_FOR_ALL_OBJECTS** следующим образом:

```
tkyte@TKYTE816> alter outline my_outline rename to plan_for_all_objects
  2  /

Outline altered.
```

Простой запрос позволяет проверить, сработало ли все, как предполагалось:

```
tkyte@TKYTE816> select name, category, sql text from user_outlines
  2  /

NAME                    CATEGORY        SQL_TEXT

PLAN_FOR_ALL_OBJECTS    MY_CATEGORY     select * from
                                        all_objects
```

Следующий шаг — изменить с помощью оператора **ALTER OUTLINE** категорию, в которой хранится шаблон. Эта версия оператора имеет следующий синтаксис:

```
alter outline иня_таблона change category to новое_имя_категории;
```

Итак, переведем наш хранимый шаблон из категории **MY_CATEGORY** в категорию **DICTIONARY_PLANS**:

```
tkyte@TKYTE816> alter outline plan_for_all_objects change category to
  2                dictionary_plans
  3  /

Outline altered.
```

```
tkyte@TKYTE816> select name, category, sql_text from user_outlines
  2  /

NAME                           CATEGORY             SQL_TEXT

PLAN_FOR_ALL_OBJECTS  DICTIONARY_PLANS       select   *   from
                                               all_objects
```

Тут тоже все понятно. Оператор **ALTER OUTLINE** просто изменяет имя категории в пользовательской схеме **OUTLN.** Чтобы продемонстрировать последний вариант использования оператора **ALTER OUTLINE,** пересоздадим план выполнения запроса в текущей среде. Синтаксис оператора в этом случае:

```
alter outline имя_таблона rebuild;
```

В настоящий момент в используемом сеансе SQL*Plus параметр **OPTIMIZER_GOAL** имеет значение **CHOOSE.** Поскольку объекты словаря не проанализированы, для запроса используется оптимизатор, основанный на правилах (если режим оптимизации — **CHOOSE** и объекты в запросе не проанализированы, используется оптимизатор, основанный на правилах). Установим цель оптимизации **ALL_ROWS,** что требует использовать оптимизатор, основанный на стоимости, и перестроим план.

```
tkyte@TKYTE816> alter session set optimizer_goal = all_rows
  2  /

Session altered.

tkyte@TKYTE816> alter outline plan for all_objects rebuild
  2  /
Outline altered.
```

Получив количество подсказок в шаблоне, можно убедиться, что сгенерированный план перестроен и отличается от исходного:

```
tkyte@TKYTE816> SELECT COUNT (*)
  2      FROM USER_OUTLINE_HINTS
  3     WHERE NAME = 'PLAN_FOR_ALL_OBJECTS'
  4  /

  COUNT(*)

       139
```

План, несомненно, отличается: теперь подсказок 139 и запрос оптимизирован в режиме **ALL_ROWS,** а не **CHOOSE.**

Оператор DROP OUTLINE

Оператор удаления шаблона — очень простой. Он имеет следующий синтаксис:

```
drop outline имя_игаблона;
```

Продолжая пример, используем этот оператор ЯОД для удаления существующего хранимого шаблона:

```
tkyte@TKYTE816> drop outline plan_for_all_objects
  2   /

Outline dropped.

tkyte@TKYTE816> select * from user_outlines;

no rows selected
```

Как вы могли убедиться, все очень просто. В следующем разделе будут описаны более гибкие процедуры для работы с группами шаблонов.

Пакет OUTLN_PKG

Перейдем теперь к пакету **OUTLN_PKG.** Этот пакет создавался:

- Для поддержки множественных операций с шаблонами, таких как удаление неиспользуемых хранимых шаблонов, удаление шаблонов определенной категории и т.д. Это можно сделать и с помощью операторов **ALTER** и **DROP,** но только по одному шаблону. Пакет **OUTLN_PKG** предлагает набор процедур для работы с несколькими шаблонами одним оператором.
- Чтобы предоставить набор процедур для утилит EXP и IMP, обеспечивающих экспорт и импорт хранимых шаблонов.

Мы опишем и продемонстрируем использование процедур пакета **OUTLN_PKG** для множественных операций. Мы не будем затрагивать процедуры пакета, предназначенные для поддержки экспорта и импорта. Эти процедуры не описаны в документации и не предназначены для вызова из другой среды, кроме утилит IMP и EXP.

Пакет **OUTLN_PKG** создается сценариями **dbmsol.sql** и **prvtol.plb,** которые находятся в каталоге **[ORACLE_HOME]/rdbms/admin.** Эти сценарии вызываются сценарием **catproc.sql** (который находится в том же каталоге) и создают пакет в базе данных по умолчанию. Помимо создания пакета **OUTLN_PKG,** эти сценарии вставляют необходимые строки в соответствующие таблицы словаря данных, чтобы зарегистрировать его функции для использования утилитами EXP/IMP. Пакет должен устанавливаться пользователем SYS или INTERNAL с помощью утилиты **SVRMGRL.** Поскольку пакет автоматически устанавливается при обновлении или установке сервера, выполнять сценарий установки вручную не понадобится.

В пакете **OUTLN_PKG** есть три процедуры, которые нас интересуют:

- **DROP_UNUSED.** Удаляет все шаблоны, в столбце **USED** которых находится значение **UNUSED**. Это хранимые шаблоны, сгенерированные, но ни разу не использовавшиеся для переписывания запроса.
- **DROP_BY_CAT.** Удаляет все шаблоны указанной категории. Если оказалось, что вся категория хранимых шаблонов больше не нужна, можно удалить их одной командой, а не выполнять оператор **DROP OUTLINE** для каждого шаблона по очереди.
- **UPDATE_BY_CAT.** Переименовывает категорию глобально, изменяя все входящие в нее шаблоны.

Процедура OUTLN_PKG.DROP_UNUSED

Эта процедура, не имеющая параметров, удаляет все не использованные шаблоны из **всех** категорий. Она находит шаблоны, в столбце **USED** для которых хранится значение UNUSED, и применяет к ним аналог оператора **DROP OUTLINE имя_шаблона.** Вот пример использования этой процедуры:

```
tkyte@TKYTE816> exec outln_pkg.drop_unused;

PL/SQL procedure successfully completed.
```

Поскольку эта процедура работает со всеми категориями, надо ее использовать осторожно. Можно ненамеренно удалить хранимый шаблон, который не следовало удалять. Этим можно свести на нет работу другого пользователя, создавшего шаблоны, но не успевшего их использовать.

Процедура OUTLN_PKG.DROP_BY_CAT

Процедура **DROP_BY_CAT** удаляет все хранимые шаблоны указанной категории. Ее можно использовать, например, при тестировании для удаления категорий шаблонов, не оправдавших ожидания. Можно также использовать эту процедуру для удаления категории шаблонов по ходу работы. Это позволяет приложению использовать планы, генерируемые оптимизатором, вместо планов, сохраненных в шаблонах. Вот простой пример использования этой процедуры:

```
tkyte@TKYTE816> select category from user_outlines;

CATEGORY

DICTIONARY_PLANS

tkyte@TKYTE816> exec outln_pkg.drop_by_cat('DICTIONARY_PLANS');

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select category from user_outlines;

no rows selected
```

Процедура OUTLN_PKG.UPDATE_BY_CAT

Эта процедура позволяет переименовать существующую категорию или объединить категории. Синтаксис вызова этой процедуры простой:

outln_pkg.update_by_cat(*старое_имя_категории*, *новое_имя_категории*) ;

Эта процедура работает следующим образом:

- Если категории **новое_имя_категории** в базе данных еще нет, все существующие шаблоны из категории **старое_имя_категории** переводятся в категорию **новое_имя_категории.**
- Если категория **новое_имя_категории** существует, все хранимые шаблоны из категории **старое_имя_категории** переносятся в категорию **новое_имя_категории.**

- Если в столбце **SQL_TEXT** хранимого шаблона в категории **старое_имя_категории** хранится текст, совпадающий с текстом одного из шаблонов в категории **новое_имя_категории,** то шаблон в новую категорию **не переносится.**

Рассмотрим пример, демонстрирующий эту возможность:

```
tkyte@TKYTE816> create outline outline_1
  2  for category CAT_1
  3  on select * from dual
  4  /

Outline created.

tkyte@TKYTE816> create outline outline_2
  2  for category CAT_2
  3  on select * from dual
  4  /
Outline created.

tkyte@TKYTE816> create outline outline_3
  2  for category CAT_2
  3  on select * from dual A
  4  /
Outline created.
```

Итак, имеется три хранимых шаблона в двух категориях. Для запроса **SELECT * FROM DUAL** есть два хранимых шаблона, а для запроса **SELECT * FROM DUAL A** — один. Посмотрим, что имеется сейчас:

```
tkyte@TKYTE816> select category, name, sql_text
  2    from user_outlines
  3   order by category, name
  4  /

CATEGORY     NAME                    SQL_TEXT

CAT_1        OUTLINE_1               select * from dual
CAT_2        OUTLINE_2               select * from dual
CAT_2        OUTLINE_3               select * from dual A
```

Как видите, в категории **CAT_1** — **1** шаблон, а в категории **CAT_2** — 2 шаблона. Более того, четко видно, что в категории **CAT_2** есть шаблон с таким же значением в столбце **SQL_TEXT,** что и в шаблоне в категории **CAT_1.** Теперь объединим категории:

```
tkyte@TKYTE816> exec outln_pkg.update_by_cat('CAT_2', 'CAT_1');

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select category, name, sql_text
  2    from user_outlines
  3   order by category, name
  4  /

CATEGORY     NAME                    SQL_TEXT

CAT_1        OUTLINE_1               select * from dual
CAT_1        OUTLINE_3               select * from dual A
CAT 2        OUTLINE 2               select * from dual
```

Как видите, шаблон из категории CAT_2 для запроса, не входящего в категорию CAT_1, был перенесен. Хранимый шаблон для дублирующегося запроса, однако, не перенесен. Дело в том, что все шаблоны должны быть уникальны по столбцу NAME и паре столбцов (CATEGORY, SIGNATURE). В пределах категории значения в столбце SQL_TEXT должны быть уникальны. Это обеспечивается путем генерации уникальной сигнатуры для значения SQL_TEXT. Если необходимо перенести шаблон OUTLINE_2 из категории CAT_2 в категорию CAT_1, придется удалить шаблон OUTLINE_1 из категории CAT_1 перед выполнением процедуры UPDATE_BY_CAT.

```
tkyte@TKYTE816> drop outline outline_1;

Outline dropped.

tkyte@TKYTE816> exec outln_pkg.update_by_cat( 'CAT_2\ 'CAT_1' );

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select category, name, sql_text
  2     from user_outlines
  3    order by category, name
  4  /

CATEGORY       NAME                      SQL_TEXT
-------------- ------------------------- --------------------
CAT_1          00TLINE_2                 select * from dual
CAT 1          OUTLINE 3                 select * from dual A
```

Проблемы

Как и при работе с любым средством, надо учитывать ряд нюансов функционирования шаблонов запросов. В этом разделе мы попытаемся их рассмотреть.

Имена шаблонов и регистр символов

В пакете **OUTLN_PKG** есть две процедуры, получающие имя категории шаблонов или имя шаблона. Поскольку параметр передается как **строка,** надо учитывать **регистр** передаваемых символов. Сервер Oracle по умолчанию хранит имена объектов в **верхнем** регистре, но если применяются идентификаторы в кавычках, можно использовать смешанный регистр. Необходимо убедиться, что регистр символов в имени категории, передаваемом процедуре **DROP_BY_CAT,** например, соответствует регистру символов имени категории, хранящегося в словаре данных. Следующий пример демонстрирует потенциальную проблему:

```
tkyte@TKYTE816> create or replace outline my_outline
  2   for category my_category
  3   on select * from dual
  4  /

Outline created.
```

```
tkyte@TKYTE816> create or replace outline my_other_outline
  2   for category "My_Category"
  3   on select * from dual
  4   /

Outline created.

tkyte@TKYTE816> select name, category, sql_text from user_outlines;

MAME                    CATEGORY          SQL_TEXT

MY_OUTLINE MY_CATEGORY              select   *   from   dual
MY_OTHER_OUTLINE        My_Category       select * from dual
```

Итак, имеется два шаблона. Обратите внимание, что имена категорий отличаются только регистром символов. Это две абсолютно разные категории. Этого удалось добиться, задав идентификатор в кавычках во втором операторе CREATE OUTLINE. Теперь кажется вполне допустимым для удаления указать имя категории в нижнем регистре, но, как будет показано ниже, это не сработает:

```
tkyte@TKYTE816> exec outln_pkg.drop_by_cat('my_category');

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select name, category, sql_text from user_outlines;

NAME                    CATEGORY          SQL_TEXT

MY_OUTLINE              MY_CATEGORY       select * from dual
MY_OTHER_OUTLINE        My_Category       select * from dual
```

Остались обе категории. Дело в том, что категории с именем в нижнем регистре нет. Теперь удалим категорию с именем в верхнем регистре:

```
tkyte@TKYTE8I6> exec outln_pkg.drop_by_cat('MY_CATEGORY');

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select name, category, sql_text from user_outlines

NAME                    CATEGORY          SQL_TEXT

MY_OTHER_OUTLINE        My_Category       select * from dual
```

И, наконец, категорию с именем в смешанном регистре:

```
tkyte@TKYTE816> exec outln_pkg.drop_by_cat('My_Category');

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select name, category, sql_text from user_outlines;

no rows selected
```

Этот побочный эффект, связанный с передачей имени объекта, а не самого объекта, иногда сбивает с толку. Подобные проблемы возникают с объектами BFILE и DIRECTORY, для которых имена тоже передаются в виде строк.

Я настоятельно не рекомендую использовать идентификаторы в кавычках. В долгосрочной перспективе они приводят к ошибкам, и на соответствующие объекты нельзя

сослаться, не указав кавычки. Я видел не одно инструментальное средство, не поддерживающее работу с идентификаторами в смешанном регистре.

Проблема с оператором ALTER SESSION

Учтите, что при отсутствии системной привилегии CREATE ANY OUTLINE, полученной непосредственно или через роль, оператор ALTER SESSION сработает без сообщений об ошибке, но шаблоны генерироваться не будут. Поэтому, если соответствующий параметр сеанса изменен, но шаблоны не генерируются, причина — в отсутствии привилегии. Необходимо получить привилегию CREATE ANY OUTLINE, непосредственно или чрез роль. Это необходимо даже в том случае, если оператор ALTER SYSTEM использовался для генерации шаблонов планов для всех сеансов. Шаблоны будут создаваться только в сеансах от имени пользователей с привилегией CREATE ANY OUTLINE.

Оператор DROP USER не удаляет шаблоны

Обычно при удалении пользователя с опцией CASCADE все принадлежащие ему объекты удаляются из базы данных. Хранимые шаблоны являются исключением из этого правила. Например:

```
sys@TKYTE816> select owner, name from dba_outlines where owner = 'TKYTE';

OWNER                          NAME
------------------------------ ------------------------------
TKYTE                          OUTLINE_1
TKYTE                          0UTLINE_2
TKYTE                          OUTLINE_3

sys@TKYTE816> drop user tkyte cascade;

User dropped.

sys@TKYTE816> select owner, name from dba_outlines where owner = 'TKYTE';

OWNER                          NAME

TKYTE                          OUTLINE_1
TKYTE                          0UTLINE_2
TKYTE                          OUTLINE_3
```

Это показывает, что даже после удаления моей учетной записи шаблоны из предыдущего примера существуют и продолжают использоваться.

Шаблоны и параметр 'CURSOR SHARING = FORCE'

В СУБД Oracle версии 8.1.6 появилась возможность, которую я называю "автоматическая подстановка связываемых переменных". В главе 10, посвященной стратегиям и средствам настройки производительности, я подчеркивал важность применения связываемых переменных и продемонстрировал новую возможность СУБД, когда ядро сер-

вера само переписывает запросы с константами так, чтобы в них использовались связываемые переменные. Этот режим — *совместное использование курсора* (cursor sharing) — имеет аномалию, проявляющуюся при работе с хранимыми шаблонами. В зависимости от того, **как** генерируется шаблон, будет запомнен план для запроса: либо со связываемыми переменными, либо без них. Пример поможет прояснить ситуацию. Выполним один и тот же запрос в сеансе с установленным параметром **CURSOR_SHARING. В** одном случае мы будем генерировать шаблон с помощью оператора **ЯОД, CREATE OUTLINE,** а в другом — автоматически, установив соответствующий параметр сеанса с помощью оператора **ALTER SESSION.** Затем сравним значения в столбце **SQL_TEXT** для полученных шаблонов:

```
tkyte@TKYTE816> alter session set cursor_sharing = force;

Session altered.

tkyte@TKYTE816> create or replace outline my_outline
  2   for category my_category
  3   on select * from dual where dummy = 'X' ;

Outline created.

tkyte@TKYTE816> alter session set create_stored_outlines = true;

Session altered.

tkyte@TKYTE816> select * from dual where dummy = 'X';

D

X

tkyte@TKYTE816> alter session set create_stored_outlines = false;

Session altered.

tkyte@TKYTE816> select name, category, sql_text from user_outlines;

NAME                            CATEGORY       SQL_TEXT

SYS_OUTLINE_0104122003150057    DEFAULT        select * from dual where dummy
                                                 = :SYS_B_0

MY_OUTLINE                      MY_CATEGORY    select * from dual where dummy
                                                 = 'X'
```

Как видите, сохраненные запросы существенно отличаются. Запрос, сгенерированный с помощью оператора CREATE **OUTLINE,** имеет в точности такой текст, как был введен. Код, соответствующий параметру **CURSOR_SHARING,** для этого запроса не выполнялся. Текст запроса был сохранен буквально. Текст же запроса для неявно сгенерированного шаблона отражает результат перезаписи. Можно явно убедиться, что константа X была автоматически заменена связываемой переменной. Этот переписанный SQL-оператор был автоматически сохранен.

В зависимости от ситуации могут пригодиться оба метода. Важно только понять, что есть существенное отличие между явно сгенерированным планом и планом, генерируемым неявно при включенном параметре **CURSOR_SHARING.**

В шаблонах используется простое сравнение текста

Механизм поиска и использования хранимых шаблонов — очень простой. Все делается на основе сравнения текста. Причем намного более простого сравнения, чем при поиске готовых планов выполнения запроса в разделяемом пуле.

При работе с шаблонами сервер Oracle ограничивается сопоставлением текста оператора. Не делается попытка проверить, что базовые объекты — одни и те же. Этим мы воспользовались в предыдущем разделе: создали схему, в ней — представления с подсказками, имена которых совпадают с именами базовых таблиц в другой схеме. Затем мы сгенерировали шаблоны для запросов к этим представлениям. На эти шаблоны существенно повлияли заданные подсказки. При выполнении точно такого же запроса в исходной схеме с таблицами (а не представлениями), сервер Oracle использовал хранимый шаблон, хотя базовые таблицы были другими. Именно так и было задумано. Намеренно было сделано так, чтобы для запросов с одинаковым текстом использовались одни и те же наборы подсказок.

Следует учитывать, что при сравнении текста требуется **полное** совпадение строк. Пробелы, символы табуляции, символы новой строки, регистр символов, — все имеет значение. Эти два запроса:

```
select * from dual;
SELECT * FROM DUAL;
```

различны с точки зрения хранимых шаблонов.

Шаблоны по умолчанию хранятся в табличном пространстве SYSTEM

По умолчанию шаблоны хранятся в табличном пространстве **SYSTEM.** Если планируется интенсивное использование хранимых шаблонов, стоит перенести их в другое табличное пространство. Как это сделать, было описано в разделе, посвященном пользователю **OUTLN.** Таблица подсказок может очень быстро разрастаться (в нашем примере для простого запроса **select * from all_objects** было сгенерировано более 100 строк в таблицу подсказок). Если разрастание табличного пространства **SYSTEM** нежелательно, рекомендуется перенести объекты пользователя **OUTLN** в другое табличное пространство.

Раскрытие условий OR

С учетом того, что механизм шаблонов запросов реализуется на основе подсказок, т.е. имеются определенные ограничения, есть один случай, мало подходящий для использования хранимых шаблонов. Речь идет о классе запросов, в которых используется *раскрытие условий OR.* При раскрытии условий OR запрос

```
select * from T where x = 5 or x = 6;
```

переписывается в виде:

```
select * from T where x = 5
Union All
select * from T where x = 6;
```

Механизм шаблонов не позволяет распространять подсказки на этот переписанный сервером план. Все хранимые подсказки будут применены к первой части запроса **UNION ALL,** но не к последующим частям. В файле **readme,** поставляемом с сервером Oracle **([ORACLE_HOME]/rdbms/doc/README.txt),** сказано:

7.4.2. Раскрытие условий OR

В планах выполнения запросов, использующих раскрытие условий **OR,** следует по возможности избегать использования хранимых шаблонов. Эта рекомендация связана как с особенностями хранимых шаблонов, использующих подсказки для изменения плана выполнения запроса, так и с особенностями реализации раскрытия условий **OR,** которые представляются в виде набора цепочек **OR,** каждая из которых представляет отдельный порядок соединения. Подсказки могут повлиять только на один порядок соединения, поскольку нет способа задать их для конкретной цепочки **OR.** Поэтому подсказки шаблона применяются к первой цепочке **OR** внутреннего представления. В итоге эти подсказки просто распространяются оптимизатором на оставшиеся цепочки **OR,** что часто приводит к неоптимальным планам выполнения, отличающимся от исходно сохраненных планов.

Решение проблемы:
Хранимые шаблоны, требующие раскрытия условий OR, можно выявить путем поиска в представлении **USER_OUTLINE_HINTS** подсказки с текстом, содержащим конструкцию **USE_CONCAT.** Выполните следующий запрос:

SELECT NAME, HINT FROM USER_OUTLINE_HINTS WHERE HINT LIKE 'USE_CONCAT%';

Любой шаблон, содержащий эту подсказку, надо либо удалить с помощью оператора **DROP OUTLINE,** либо перенести в неиспользуемую категорию с помощью следующей команды:

ALTER OUTLINE <имя_шаблона> CHANGE CATEGORY TO <имя_неиспользуемой_категории>;

Производительность

Очевидный вопрос: как использование шаблонов влияет на производительность во время выполнения? Ответ: незначительно. При использовании шаблонов на этапе разбора запроса дополнительно расходуется незначительное количество ресурсов, в основном — при первоначальной генерации и сохранении плана выполнения запроса (как и следовало ожидать).

Чтобы проверить это, я создал небольшой блок PL/SQL, который разбирает, выполняет и извлекает строки результатов для x простых запросов **(select * from T1,** где **Tl —**

таблица из одной строки и одного столбца). При этом основное время уходит на разбор запросов. Для этого я выполнил следующий блок, создающий 100 таблиц:

```
tkyte@TKYTE816> begin
  2      for i in 1 .. 100 loop
  3              begin
  4                    execute immediate 'drop table t'||i;
  5              exception
  6                    when others then null;
  7              end;
  8             execute immediate 'create table t'||i||' (dummy char(1))';
  9            execute immediate 'insert into t' ||i||' values ( "x" ) • ;
 10      end loop;
 11   end;
 12   /

PL/SQL procedure successfully completed.
```

Итак, после создания 100 таблиц с именами от T1 до T100 я выполнил блок кода, по сути только разбирающий SQL-оператор и помещающий его в разделяемый пул. Мы хотим оценить влияние шаблонов, а не разбора запроса:

```
tkyte@TKYTE816> declare l_tmp char(1); l_start number :=
                                              dbms_utility.get_time; begin
  2   select * into l_tmp from t1;
  3   select * into l_tmp from t2;
  4   select * into l_tmp from t3;
...   ...
...   ...
...   ...
 99   select * into l_tmp from t98;
100   select * into l_tmp from t99;
101   select * into l_tmp from t100;
102  dbms_output.put_line(round((dbms_utility.get_time-l_start)/100,
                                                    2 ) || ' seconds');
103   end;
104   /
.89 seconds
```

После заполнения кэша я выполнил блок еще пару раз, чтобы увидеть, как долго он будет выполняться:

```
tkyte@TKYTE816> declare l_tmp char(1); l_start number :=
                                              dbms_utility.get_time; begin
  2   select * into l_tmp from t1;
  3   select * into l_tmp from t2;
  4   select * into l_tmp from t3;
...   ...
...   ...
...   ...
 99   select * into l_tmp from t98;
100   select * into l_tnp from t99;
101   select * into l_tmp from t100;
```

```
102   dbms_output.put_line(round((dbms_utility.gat time-1 start)/100,
                                                          2 ) | | ' seconds');
103   end;
104   /
.02 seconds
```

Он стабильно выполнялся примерно за 0,02 секунды. Потом я включил создание шаблонов:

```
tkyte@TKYTE816> alter session set create_stored_outlines = testing;

Session altered.

tkyte@TKYTE816> declare l_tmp char(1); l_start number :=
dbms_utility.get_time; begin
   2   select * into l_tmp from t1;
   3   select * into l_tmp from t2;
   4   select * into l_tmp from t3;
...   ...
...   ...
...   ...
  99   select * into l_tmp from t98;
100   select * into l_tmp from t99;
101   select * into l_tmp from t100;
102  dbms_output.put_line(round((dbms_utility.get_time-l start)/100, 2) ||'
seconds');
103   end;
104   /
.82 seconds
```

Первый раз, когда сохранялись шаблоны, выполнение продолжалось примерно 0,82 секунды. Потребовалось примерно столько же времени, как и для первоначального разбора запросов. После этого оказалось, что последующие выполнения продолжались примерно 0,02 секунды. После замедления, связанного с начальным запоминанием шаблонов, время выполнения стало таким же, как и до включения запоминания шаблонов. В загруженной многопользовательской среде результаты могут быть другими, и для достижения удовлетворительной производительности при работе с таблицами **OUTLN** может понадобиться настройка параметров (например, добавление списков свободных мест), чтобы принять большое количество одновременно вставляемых строк.

При этом надо учесть следующее. Работать постоянно с установленным параметром **CREATE_STORED_OUTLINES = TRUE** не надо. Он устанавливается на некоторый период времени, чтобы перехватить интересующие вас запросы и планы их выполнения. В производственной среде обычно устанавливается параметр **USE_STORED_OUTLINES = TRUE,** а не **CREATE.** Идея в том, что даже если расходы ресурсов на генерацию планов будут существенными, это не будет делаться в производственной среде. Дополнительный расход ресурсов произойдет только в среде разработки или тестирования.

Теперь давайте оценим расход ресурсов на фактическое **использование** хранимых планов для этих простых запросов:

```
tkyte@TKYTE816> alter session set use_stored_outlines=testing;

Session altered.
```

```
tkyte@TKYTE816> select used, count(*) from user_outlines group by used;

USED       COUNT(*)

UNUSED          100

tkyte@TKYTE816> declare l_tmp char(1); l_start number :=
                                                dbms_utility.get_time; begin
  2   select * into l_tmp from t1;
  3   select * into l_tmp from t2;
  4   select * into l_tmp from t3;
...   ...
...   ...
...   ...
 99   select * into l_tmp from t98;
100   select * into l_tmp from t99;
101   select * into l_tmp from t100;
102   dbms_output.put_line(round((dbms_utility.get_time-l_start)/100,
                                                          2)||' seconds');
103   end;
104   /
.32 seconds

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select used, count(*) from user_outlines group by used;

USED       COUNT(*)
---------- ------------
USED            100

tkyte@TKYTE816> declare l_tmp char(1); l_start number :=
                                                dbms_utility.get_time; begin
  2   select * into l_tmp from t1;
  3   select * into l_tmp from t2;
  4   select * into l_tmp from t3;
...   ...
...   ...
...   ...
 99   select * into l_tmp from t98;
100   select * into l_tmp from t99;
101   select * into l_tmp from tlOO;
102   dbms_output.put_line(round((dbms_utility.get_time-l_start)/100,
                                                          2>||'  seconds');
103   end;
104   /
.03 seconds

PL/SQL procedure successfully completed.

tkyte@TKYTE816> declare l_tmp char(1); l_start number :=
                                                dbms_utility.get_time; begin
  2   select * into l_tmp from t1;
  3   select * into l_tmp from t2;
  4   select * into l_tmp from t3;
```

```
  99   select * into l_tmp from t98;
 100   select * into l_tmp from t99;
 101   select * into l_tmp from t100;
 102  dbms_output.put_line(round((dbms_utility.get_time-l_start)/100,
                                                              2)||' seconds');
 103   end;
 104   /
.03 seconds

PL/SQL procedure successfully completed.
```

Первый раз, когда пришлось повторно разбирать эти запросы после начала использования хранимых шаблонов, на их выполнение ушло 0,32 секунды. Если сравнить это значение с временем, затраченным на начальный разбор без использования хранимых шаблонов (без их сохранения или использования), оказывается, что время разбора существенно не изменилось. Мы также убедились, что хранимые шаблоны используются, поскольку в столбце USED после выполнения блока для всех строк вместо значения UNUSED появилось значение USED. А это признак того, что в запросах использовались подсказки. Последующее повторное выполнение этого же блока показало, что благодаря добавлению и сохранению подсказок в разделяемом пуле производительность при использовании хранимых шаблонов не снижается.

Итак, использование хранимых шаблонов в приложении существенно не влияет на производительность разбора во время выполнения после первоначального разбора и помещения операторов в разделяемый пул. Использование хранимых шаблонов повышает производительность в той степени, насколько оптимален план выполнения запроса; изменение же хранимыми шаблонами текста запроса заметно на производительности не сказывается.

Пространство имен шаблонов - глобально

Казалось бы, что имена шаблонов, подобно другим объектам базы данных, например таблицам, должны быть уникальны в схеме пользователя-создателя. Тем не менее это не так. Имя шаблона должно быть уникальным в базе данных, аналогично имени табличного пространства или каталога. Наличие столбца OWNER в представлении USER_OUTLINES сбивает с толку, ведь фактически шаблоном никто не владеет. В столбце OWNER указано имя пользователя, создавшего шаблон.

В этом можно убедиться с помощью простого теста:

```
tkyte@TKYTE816> create outline the_outline
  2   on select * from dual;

Outline created.

tkyte@TKYTE816> connect system

system@TKYTE816> select owner, name from dba_outlines;
```

```
OWNER                          NAME
------------------------------ --------------------------------

TKYTE                          THE_OUTLINE

system@TKYTE816> create outline the_outline
  2  on select * from dual;
on select * from dual

ERROR at line 2:

ORA-18004: outline already exists

system@TKYTE816> drop outline the_outline;

Outline dropped.

system@TKYTE816> select owner, name from dba_outlines;

no rows selected
```

Итак, как видите, пользователь **SYSTEM** не может создать второй шаблон с именем **THE_OUTLINE,** пока не использует оператор **CREATE OR REPLACE** (который перепишет существующий шаблон) или не удалит уже существующий шаблон. (Обратите внимание: уточнять имя шаблона именем схемы, **ВЛАДЕЛЕЦ.ИМЯ_ШАБЛОНА,** как в случае других объектов, нет необходимости).

Это надо учитывать, чтобы случайно не переписать чужой шаблон. При неявном создании шаблонов путем установки параметра **ALTER SESSION SET CREATE_STORED_OUTLINES** эта проблема не возникает, поскольку уникальное имя шаблона будет сгенерировано сервером. Проблема может возникнуть при явном создании и именовании шаблонов.

Ошибки, которые можно допустить

В этом разделе перечислены ошибки, которые можно допустить при работе с шаблонами.

ORA-18001 "не указаны операторы для ALTER OUTLINE"

```
// * Причина: При разборе оказалось, что не указана необходимая
//            конструкция в команде
// * Действие: Повторно выполните оператор, указав все необходимые
//             конструкции ALTER OUTLINE.
```

Это сообщение об ошибке выдается только при неправильном использовании оператора **ALTER OUTLINE.** Например:

```
ops$tkyte@DEV816> alter outline xxxx
  2  /
alter outline xxxx
              *
ERROR at line 1:
ORA-18001: no options specified for ALTER OUTLINE
```

Текст сообщения об ошибке первоначально приведен так, как он выдается СУБД Oracle версии 8.1.6.0.0 при установке русского языка для сообщений. Обратите внимание на несоответствие терминологии, предлагаемой компанией Oracle. В примерах оставлены сообщения на английском языке. - ***Прим. научи, ред.***

Решение простое: укажите одну из трех допустимых опций (**RENAME**, **REBUILD**, **CHANGE**) оператора и выполните его повторно. Подробнее об этом см. в разделе "Управление шаблонами".

ORA-18002 "указанный вариант не существует"

```
// * Причина: Шаблон либо не существует, либо был удален или изменен
//            другим потоком.
// * Действие:
```

Это сообщение об ошибке тоже вполне очевидно. Указанного шаблона больше нет: либо его никогда не было, либо кто-то его удалил.

ORA-18003 "вариант с данной сигнатурой уже существует"

```
// * Причина: Алгоритм генерации сигнатур создает сигнатуры длиной 16 байт,
//            так что совпадение сигнатур весьма маловероятно. Это сообщение
//            выдается в случае такого совпадения.
// * Действие: Выполните повторно оператор, который привел к созданию
//             шаблона, добавив в текст пробел, или отнесите шаблон к
//             другой категории.
```

Мне не удалось придумать тестовый пример для этого сообщения об ошибке; если вы его получили, значит, вам сильно не повезло. Сигнатуры запросов позволяют быстро их находить. Поскольку запросы могут быть очень длинными, для поиска используются числовые сигнатуры.

ORA-18004 "вариант уже существует"

```
II * Причина: Шаблон с указанным именем или для данного SQL-оператора уже
//            существует.
// * Действие:
```

Это сообщение об ошибке самоочевидно. Вы попытались создать шаблон с явно указанным именем, но шаблон с таким именем уже существует. Можно сделать следующее:

- выбрать другое имя;
- использовать оператор CREATE OR REPLACE и переписать существующий шаблон;
- удалить существующий шаблон, а затем создать новый с таким же именем.

ORA-18005-18007

Эти три сообщения об ошибках тесно взаимосвязаны, поэтому я опишу причины всех трех вместе:

- ORA-18005 "для этой операции требуется привилегия CREATE ANY OUTLINE".
- ORA-18006 "для этой операции требуется привилегия DROP ANY OUTLINE".
- ORA-18007 "для этой операции требуется привилегия ALTER ANY OUTLINE ".

Эти сообщения об ошибках выдаются при попытке выполнить с шаблоном операцию, на выполнение которой нет соответствующей привилегии. Это может казаться

странным, особенно при работе с собственными шаблонами. Как объяснялось в разделе "Проблемы", шаблоны никому не принадлежат; их имена — глобальны (как имена табличных пространств). Поэтому пользователь может создать шаблон с помощью CREATE, но затем не сможет удалить или изменить его. Другой пользователь может изменять шаблоны, но не может создавать или удалять их и т.д.

Подробнее о необходимых привилегиях см. в разделе "Привилегии, необходимые для работы с хранимыми шаблонами".

Резюме

В этой главе подробно рассмотрена возможность стабилизации плана оптимизатора, поддерживаемая в СУБД Oracle8i. Эта возможность позволяет стабилизировать производительность SQL-операторов, независимо от изменений в базе данных (например, обновления версии сервера, изменения параметров инициализации и т.д.). Был представлен ряд других полезных вариантов использования этой возможности, например, для настройки производительности приложений, которые по любой причине нельзя изменять, для поиска используемых индексов, получения списка выполняемых SQL-операторов и т.п. Использование хранимых шаблонов не требует изменения приложения и влечет минимальные дополнительные расходы ресурсов во время выполнения. Надо учитывать ряд проблем, связанных с использованием хранимых шаблонов; но если помнить об этом, хранимые шаблоны сослужат хорошую службу.

Предметный указатель

H

O

П

P

С

T

12

Аналитические функции

SQL — очень мощный язык и лишь очень немногие запросы в нем нельзя создать. По опыту знаю, что можно придумать хитрый SQL-запрос для получения ответа практически на любой вопрос относительно любых данных. Однако производительность некоторых из этих запросов крайне низкая, да и придумать их непросто. Ряд запросов, которые сложно сформулировать на обычном языке SQL, весьма типичны:

- **Подсчет промежуточной суммы.** Показать суммарную зарплату сотрудников отдела построчно, чтобы в каждой строке выдавалась сумма зарплат всех сотрудников вплоть до указанного.
- **Подсчет процентов в группе.** Показать, какой процент от общей зарплаты по отделу составляет зарплата каждого сотрудника. Берем его зарплату и делим на сумму зарплат по отделу.
- **Запросы первых N.** Найти N сотрудников с наибольшими зарплатами или N наиболее продаваемых товаров по регионам.
- **Подсчет скользящего среднего.** Получить среднее значение по текущей и предыдущим N строкам.
- **Выполнение ранжирующих запросов.** Показать относительный ранг зарплаты сотрудника среди других сотрудников того же отдела.

Аналитические функции, появившиеся в версии Oracle 8.1.6, создавались для решения именно этих задач. **Они** расширяют язык SQL так, что подобные операции не только проще записываются, но и быстрее выполняются по сравнению с использованием чистого языка SQL. Эти расширения сейчас изучаются комитетом ANSI SQL с целью включения в спецификацию языка SQL.

Мы начнем эту главу с примера, демонстрирующего возможности аналитических функций. После этого будет представлен полный синтаксис и описание всех функций, а также ряд практических примеров выполнения перечисленных выше операций. Как обычно, в конце будут рассмотрены потенциальные проблемы использования аналитических функций.

Пример

Простой пример подсчета промежуточной суммы зарплат по отделам с описанием того, что же в действительности происходит, позволят получить начальное представление о принципах использования аналитических функций:

```
tkyte@TKYTE816> break on deptno skip 1

tkyte@TKYTE816> select ename, deptno, sal,
  2       sum(sal) over
  3          (order by deptno, ename) running_total,
  4     sum(sal)    over
  5          (partition by deptno
  6         order by ename) department_total,
  7       row_number() over
  8          (partition by deptno
  9          order by ename) seq
 10  from emp
 11  order by deptno, ename
 12  /
```

ENAME	DEPTNO	SAL	RUNNING_TOTAL	DEPARTMENT_TOTAL	SEQ
CLARK	10	2450	2450	2450	1
KING		5000	7450	7450	2
MILLER		1300	8750	8750	3
ADAMS	20	1100	9850	1100	1
FORD		3000	12850	4100	2
JONES		2975	15825	7075	3
SCOTT		3000	18825	10075	4
SMITH		800	19625	10875	5
ALLEN	30	1600	21225	1600	1
BLAKE		2850	24075	4450	2
JAMES		950	25025	5400	3
MARTIN		1250	26275	6650	4
TURNER		1500	27775	8150	5
WARD		1250	29025	9400	6

```
14 rows selected.
```

В представленном выше коде удалось получить значение RUNNING_TOTAL для запроса в целом. Это было сделано по всему упорядоченному результирующему множеству с помощью конструкции SUM(SAL) OVER (ORDER BY DEPTNO, ENAME). Также удалось подсчитать промежуточные суммы по отделам, сбрасывая их в ноль при переходе к следующему отделу. Этого удалось добиться благодаря конструкции PARTITION BY

DEPTNO в **SUM(SAL)** — в запросе была указана конструкция, задающая условие разбиения данных на группы. Для последовательной нумерации строк в каждой группе, в соответствии с критериями упорядочения, использовалась функция **ROW_NUMBER()** (для выдачи этого номера строки был добавлен столбец **SEQ**). В результате видно, что **SCOTT** — четвертый по списку сотрудник в отделе 20 при упорядочении по фамилии (**ENAME**). Функция **ROW_NUMBER()** используется и во многих других ситуациях, например для транспонирования или преобразования результирующих множеств (как будет описано далее).

Этот новый набор функциональных возможностей содержит много замечательного. Он открывает абсолютно новые перспективы работы с данными. Можно избавиться от большого объема процедурного кода и сложных (или неэффективных) запросов, требующих много времени на разработку, и получить при этом желаемый результат. Чтобы почувствовать, насколько эффективными могут быть аналитические функции по сравнению с "чисто реляционными способами", давайте оценим производительность в случае 1000 строк, а не 14. При этом сравним производительность двух запросов: с новыми аналитическими функциями и на основе "старых" реляционных методов. Следующая пара операторов позволит создать аналог таблицы **SCOTT.EMP** с тремя столбцами — **ENAME**, **DEPTNO** и **SAL** — и индексом (единственным необходимым в данном примере). Я буду выбирать данные по столбцам **DEPTNO** и **ENAME**:

```
tkyte@TKYTE816>  create table t
  2  as
  3 select object_name ename,
  4       mod(object_id,50) deptno,
  5        object_id sal
  6  from all_objects
  7  where rownum <= 1000
  8  /

Table created.

tkyte@TKYTE816> create index t_idx on t(deptno,ename);
Index created.
```

Повторим запрос, но к новой таблице, задав установку **AUTOTRACE TRACEONLY**, чтобы увидеть, сколько и чего пришлось делать (для этого необходимо наличие роли **PLUSTRACE**):

```
tkyte@TKYTE816> set autotrace traceonly

tkyte@TKYTE816> select ename, deptno, sal,
  2    sum(sal) over
  3      (order by deptno, ename) running_total,
  4    sum(sal) over
  5       (partition by deptno
  6       order by ename) department_total,
  7    row_number() over
  8       (partition by deptno
  9       order by ename) seq
 10  from t emp
 11 order by deptno, ename101
```

```
 12  /

1000 rows selected.

Elapsed:  00:00:00.61

Execution  Plan

   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   WINDOW  (BUFFER)
   2    1     TABLE ACCESS  (BY INDEX ROWID)  OF  'T'
   3    2       INDEX  (FULL SCAN)  OF  'T_IDX'  (NON-UNIQUE)

Statistics

          0  recursive calls
          2  db block gets
        292  consistent gets
         66  physical reads
          0  redo size
     106978  bytes sent via SQL*Net to client
       7750  bytes received via SQL*Net  from client
         68  SQL*Net roundtrips to/from client
          0  sorts  (memory)
          1  sorts  (disk)
       1000  rows processed
```

Итак, потребовалось 0,61 секунды и 294 логические операции ввода-вывода. Теперь выполним эквивалентный запрос, но используя только "стандартные" возможности языка SQL:

```
tkyte@TKYTE816>  select  ename,  deptno,  sal,
  2     (select sum(sal)
  3       from t e2
  4       where e2.deptno < emp.deptno
  5       or  (e2.deptno = emp.deptno and e2.ename <= emp.ename ))
  6   running_total,
  7     (select sum(sal)
  8       from t e3
  9       where e3.deptno = emp.deptno
 10       and e3.ename <= emp.ename)
 11   department_total,
 12     (select count(ename)
 13       from t e3
 14       where e3.deptno = emp.deptno
 15       and e3.ename <= emp.ename)
 16   seq
 17   from t emp
 18   order by deptno, ename
 19   /

1000 rows selected.

Elapsed: 00:00:06.89
```

```
Execution Plan

   0     SELECT STATEMENT Optimizer=CHOOSE
   1    0  TABLE ACCESS (BY INDEX ROWID) OF 'T'
   2    1    INDEX (PULL SCAN) OF 'T_IDX' (NON-UNIQUE)

Statistics

         0  recursive calls
         0  db block gets
    665490  consistent gets
         0  physical reads
         0  redo size
    106978  bytes sent via SQL*Net to client
      7750  bytes received via SQL*Net from client
        68  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
      1000  rows processed

tkyte@TKYTE816> set autotrace off
```

Оба запроса дали одинаковые результаты, но производительность отличается существенно. Время выполнения больше во много раз, а количество логических операций ввода-вывода увеличилось на несколько порядков. Аналитические функции обработали результирующее множество, использовав намного меньше ресурсов и, соответственно, быстрее. Более того, если рассмотреть синтаксис аналитических функций, оказывается, что записывать с их помощью запросы намного проще, чем на стандартном языке SQL. Чтобы почувствовать разницу, сравните текст двух представленных выше запросов.

Как работают аналитические функции

В первой части этого раздела будут представлены подробности синтаксиса и определены термины. После этого мы перейдем непосредственно к примерам. Я продемонстрирую многие из 26 новых функций (не все, потому что при этом некоторые примеры повторялись бы). Аналитические функции используют общий синтаксис и предоставляют специфические возможности, создававшиеся для нужд технических дисциплин, незнакомых большинству разработчиков. Поняв принцип написания аналитических функций — как фрагментировать данные, как задавать окна данных и так далее, — использовать эти функции будет очень легко.

Синтаксис

Синтаксис вызова аналитической функции на вид весьма прост, но эта простота может быть обманчивой. Все начинается с такой конструкции:

```
ИМЯ_ФУНКЦИИ(<аргумент>,< аргумент >, ...)
OVER
 (<конструкция_фрагментации> <конструкция_упорядочения> <конструкция_окна>)
```

Вызов аналитической функции может содержать до четырех частей: аргументы, конструкция фрагментации, конструкция упорядочения и конструкция, задающая окно. В представленном выше примере:

```
4       sum(sal) over
5   (partition    by     deptno
6           order by ename) department_total,
```

- **SUM** — имя функции.
- **(SAL)** — аргумент аналитической функции. Аналитические функции принимают от нуля до трех аргументов. В качестве аргументов передаются выражения, т.е. вполне можно было бы использовать **SUM(SAL+COMM).**
- **OVER** — ключевое слово, идентифицирующее эту функцию как аналитическую. В противном случае синтаксический анализатор не мог бы отличить функцию агрегирования SUM() от аналитической функции SUM(). Конструкция после ключевого слова **OVER** описывает срез данных, "по которому" будет вычисляться аналитическая функция.
- **PARTITION** BY **DEPTNO** — необязательная конструкция фрагментации. Если конструкция фрагментации не задана, все результирующее множество считается одним большим фрагментом. Это используется для разбиения результирующего множества на группы, так что аналитическая функция применяется к группам, а не ко всему результирующему множеству. В первом примере главы, когда конструкция фрагментации не указывалась, функция **SUM** по столбцу **SAL** вычислялась для всего результирующего множества. Фрагментируя результирующее множество по столбцу **DEPTNO,** мы вычисляли **SUM** по столбцу **SAL** для каждого отдела **(DEPTNO),** сбрасывая промежуточную сумму для каждой группы.
- **ORDER BY ENAME** — необязательная конструкция **ORDER BY;** для некоторых функций она обязательна, для других — нет. Функции, зависящие от упорядочения данных, например **LAG** и **LEAD,** которые позволяют обратиться к предыдущим и следующим строкам в результирующем множестве, требуют обязательного указания конструкции **ORDER BY.** Другие функции, например **AVG,** не требуют. Эта конструкция обязательна, если используется любая функция работы с окном (подробнее см. далее в разделе "Конструкция окна"). Конструкция **ORDER BY** определяет, как упорядочиваются данные в группах при вычислении аналитической функции. В нашем случае упорядочивать по **DEPTNO** и **ENAME** не нужно, потому что по столбцу **DEPTNO** выполнялась фрагментация, т.е. неявно предполагается, что столбцы, по которым выполняется фрагментация, по определению входят в ключ сортировки (конструкция **ORDER BY** применяется к каждому фрагменту поочередно).
- **Конструкция окна в данном примере отсутствует.** Именно ее синтаксис иногда кажется сложным. Подробно возможные способы задания конструкции окна будут рассмотрены ниже.

Теперь более детально рассмотрим каждую из четырех частей вызова аналитической функции, чтобы понять, как их можно задавать.

Функции

Сервер Oracle предлагает 26 аналитических функций. Они разбиваются на четыре основных класса по возможностям.

Первый класс образуют различные функции *ранжирования,* позволяющие строить запросы типа "первых N". Мы уже использовали одну функцию этого класса, **ROW_NUMBER**, при генерации столбца **SEQ** в предыдущем примере. Она ранжировала сотрудников в отделах по фамилии (**ENAME**). Точно так же их можно было бы ранжировать по зарплате (**SALARY**) или любому другому атрибуту.

Второй класс образуют *оконные* функции, позволяющие вычислять разнообразные агрегаты. В первом примере этой главы была показана такая функция — мы вычисляли **SUM(SAL)** по разным группам. Вместо функции **SUM** можно было использовать и другие функции агрегирования, например **COUNT, AVG, MIN, MAX** и т.д.

К третьему классу относятся различные *итоговые* функции. Они очень похожи на оконные, поэтому имеют те же имена: **SUM**, **MIN**, **MAX** и т.д. Тогда как оконные функции используются для работы с окнами данных, как промежуточная сумма в предыдущем примере, итоговые функции работают со всеми строками фрагмента или группы. Например, если бы в первоначальном запросе использовались обращения:

```
sum(sal) over () total_salary,
sum(sal) over (partition by deptno) total_salary_for_department
```

мы бы получили общие суммы по группам, а не промежуточные. Ключевое отличие итоговой функции от оконной — отсутствие конструкции **ORDER BY** в операторе **OVER**. При отсутствии конструкции **ORDER BY** функция применяется к каждой строке группы. При наличии конструкции **ORDER BY** функция применяется к окну (подробнее об этом в разделе, описывающем конструкцию окна).

Есть также функции **LAG** и **LEAD**, позволяющие получать значения из предыдущих или следующих строк результирующего множества. Это помогает избежать самосоединения данных. Например, если в таблице записаны даты визитов пациентов к врачу и необходимо вычислить время между визитами для каждого их них, очень пригодится функция **LAG**. Можно просто фрагментировать данные по пациентам и отсортировать их по дате. После этого функция **LAG** легко сможет вернуть данные предыдущей записи для пациента. Останется вычесть из одной даты другую. До появления аналитических функций для получения этих данных приходилось организовывать сложное соединение таблицы с ней же самой.

Наконец, есть большой класс *статистических* функций, таких как **VAR_POP**, **VAR_SAMP**, **STDEV_POP**, набор функций линейной регрессии и т.п. Эти функции позволяют вычислять значения статистических показателей для любого неупорядоченного фрагмента.

В конце раздела, посвященного синтаксису, представлена таблица с кратким объяснением назначения всех аналитических функций.

Конструкция фрагментации

Конструкция **PARTITION BY** логически разбивает результирующее множество на N групп по критериям, задаваемым выражениями фрагментации. Слова "фрагмент" и "груп-

па" в этой главе и в документации Oracle используются как синонимы. Аналитические функции применяются к каждой группе независимо, — для каждой новой группы они сбрасываются. Например, ранее при демонстрации функции, вычисляющей промежуточную сумму зарплат, фрагментация выполнялась по столбцу **DEPTNO.** Когда значение в столбце **DEPTNO** в результирующем множестве изменялось, происходил сброс промежуточной суммы в ноль, и суммирование начиналось заново.

Если не указать конструкцию фрагментации, все результирующее множество считается одной группой. Во первом примере мы использовали функцию **SUM(SAL)** без конструкции фрагментации, чтобы получить промежуточные суммы для всего результирующего множества.

Интересно отметить, что каждая аналитическая функция в запросе может иметь уникальную конструкцию фрагментации; фактически уже в простейшем примере в начале главы это и было сделано. Для столбца **RUNNING_TOTAL** конструкция фрагментации не была задана, поэтому целевой группой было все результирующее множество. Для столбца **DEPARTMENTAL_TOTAL** результирующее множество фрагментируется по отделам, что позволило вычислять промежуточные суммы для каждого из них.

Синтаксис конструкции фрагментации прост и очень похож на синтаксис конструкции **GROUP BY** в обычных SQL-запросах:

PARTITION BY выражение [, выражение] [, выражение]

Конструкция упорядочения

Конструкция **ORDER BY** задает критерий сортировки данных в каждой группе (в каждом фрагменте). Это, несомненно, влияет на результат выполнения любой аналитической функции. При наличии (или отсутствии) конструкции **ORDER BY** аналитические функции вычисляются по-другому. В качестве примера рассмотрим, что происходит при использовании функции **AVG()** с конструкцией **ORDER BY** и без оной:

```
scott@TKYTE816> select ename, sal, avg(sal) over ()
  2  from emp;
  3  /

ENAME             SAL AVG(SAL)OVER()

SMITH          800.00       2073.21
ALLEN         1600.00       2073.21
WARD          1250.00       2073.21
JONES         2975.00       2073.21
MARTIN        1250.00       2073.21
BLAKE         2850.00       2073.21
CLARK         2450.00       2073.21
SCOTT         3000.00       2073.21
KING          5000.00       2073.21
TURNER        1500.00       2073.21
ADAMS         1100.00       2073.21
JAMES          950.00       2073.21
FORD          3000.00       2073.21
MILLER        1300.00       2073.21

14 rows selected.
```

```
scott@TKYTE816>select ename, sal, avg(sal) over (ORDER BY ENAME)
  2  from emp
  3  order by ename
  4  /

ENAME          SAL AVG(SAL)OVER(ORDERBYENAME)

ADAMS       1100.00                  1100.00
ALLEN       1600.00                  1350.00
BLAKE       2850.00                  1850.00
CLARK       2450.00                  2000.00
FORD        3000.00                  2200.00
JAMES        950.00                  1991.67
JONES       2975.00                  2132.14
KING        5000.00                  2490.63
MARTIN      1250.00                  2352.78
MILLER      1300.00                  2247.50
SCOTT       3000.00                  2315.91
SMITH        800.00                  2189.58
TURNER      1500.00                  2136.54
WARD        1250.00                  2073.21

14 rows selected.
```

В отсутствие конструкции **ORDER BY** среднее значение вычисляется по всей группе, и одно и то же значение выдается для каждой строки (функция используется как итоговая). Когда функция AVG() используется с конструкцией **ORDER BY**, среднее значение в каждой строке является средним по текущей и всем предыдущим строкам (функция используется как оконная). Например, средняя зарплата для пользователя **ALLEN** в результатах выполнения запроса с конструкцией **ORDER BY** — 1350 (среднее для значений 1100 и 1600).

*Немного забегая вперед, в следующий раздел, посвященный конструкции окна, можно сказать, что наличие конструкции **ORDER BY** в вызове аналитической функции добавляет стандартную конструкцию окна — **RANGE UNBOUNDED PRECEDING**. Это означает, что для вычисления используется набор из всех предыдущих и текущей строки в текущем фрагменте. При отсутствии **ORDER BY** стандартным окном является весь фрагмент.*

Чтобы реально почувствовать, как все это работает, рекомендую применить одну и ту же аналитическую функцию при двух различных конструкциях **ORDER BY**. В первом примере текущая сумма вычисляется для всей таблицы **EMP** с использованием конструкции **ORDER BY DEPTNO, ENAME**. При этом текущая сумма вычисляется для всех строк, причем, порядок их просмотра определяется конструкцией **ORDER BY**. Если изменить порядок указания столбцов в этой конструкции на противоположный или вообще сортировать по другим столбцам, получаемые текущие суммы будут существенно отличаться; общая сумма в последней строке совпадет, но все промежуточные значения будут другими. Например:

```
ops$tkyte@DEV816> select ename, deptno,
  2    sum(sal) over (order by ename, deptno) sum_ename_deptno,
  3    sum(sal) over (order by deptno, ename) sum_deptno_ename
  4  from emp
  5  order by ename, deptno
  6  /

ENAME          DEPTNO SUM_ENAME_DEPTNO SUM_DEPTNO_ENAME

ADAMS              20             1100             9850
ALLEN              30             2700            21225
BLAKE              30             5550            24075
CLARK              10             8000             2450
FORD               20            11000            12850
JAMES              30            11950            25025
JONES              20            14925            15825
KING               10            19925             7450
MARTIN             30            21175            26275
MILLER             10            22475             8750
SCOTT              20            25475            18825
SMITH              20            26275            19625
TURNER             30            27775            27775
WARD               30            29025            29025

14 rows selected.
```

Оба столбца **SUM(SAL)** одинаково корректны; один из них содержит **SUM(SAL)** при упорядочении по столбцу **DEPTNO**, а потом — по **ENAME**, а другой — при упорядочении по столбцу **ENAME**, а потом — по **DEPTNO**. Поскольку результирующее множество упорядочено по (**ENAME**, **DEPTNO**), значения **SUM(SAL)**, вычислявшиеся именно в этом порядке, кажутся более корректными, но общая сумма совпадает: **29025**.

Конструкция **ORDER BY** в аналитических функциях имеет следующий синтаксис:

```
ORDER BY выражение [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Она совпадает с конструкцией **ORDER BY** для запроса, но будет упорядочивать строки только в пределах фрагментов и может не совпадать с конструкцией **ORDER BY** для запроса в целом (или любого другого фрагмента). Конструкции **NULLS FIRST** и **NULLS LAST** впервые появились в версии Oracle 8.1.6. Они позволяют указать, где при упорядочении должны быть значения **NULL** — в начале или в конце. В случае сортировки по убыванию (**DESC**), особенно в аналитических функциях, эта новая возможность принципиально важна. Почему — описано в разделе "Проблемы" в конце главы.

Конструкция окна

Синтаксис этой конструкции на первый взгляд кажется сложным из-за используемых ключевых слов. Конструкция вида **RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW**, задающая стандартное окно при использовании конструкции **ORDER BY**, не похожа на те, что постоянно используются разработчиками. Синтаксис конструкции окна достаточно сложен для описания. Вместо попыток перерисовать синтаксические схемы, представленные в руководстве *Oracle8i SQL Reference Manual,* я перечислю все варианты конструкции окна и опишу, какой набор данных в пределах груп-

пы задает соответствующий вариант. Сначала, однако, давайте разберемся, что вообще позволяет сделать конструкция окна.

Конструкция окна позволяет задать перемещающееся или жестко привязанное окно (набор) данных в пределах группы, с которым будет работать аналитическая функция. Например, конструкция диапазона **RANGE UNBOUNDED PRECEDING** означает: "применять аналитическую функцию к каждой строке данной группы, с первой по текущую". Стандартным является жестко привязанное окно, начинающееся с первой строки группы и продолжающееся до текущей. Если используется следующая аналитическая функция:

```
SUM(sal) OVER
  (PARTITION BY deptno
   ORDER BY ename
   ROWS 2 PRECEDING) department_total2,
```

то будет создано перемещающееся окно в группе, и сумма зарплат будет вычисляться по столбцу SAL текущей и двух предыдущих строк в этой группе. Если необходимо создать отчет, показывающий сумму зарплат текущего и двух предыдущих сотрудников отдела, соответствующий сценарий может выглядеть так:

```
scott@TKYTE816> break on deptno

scott@TKYTE816> select deptno, ename, sal,
  2   sum(sal) over
  3     (partition by deptno
  4      order by ename
  5     rows 2 preceding) sliding_total
  6  from emp
  7  order by deptno, ename
  8  /

    DEPTNO ENAME             SAL SLIDING_TOTAL

        10 CLARK            2450          2450
           KING             5000          7450
           MILLER           1300          8750

        20 ADAMS            1100          1100
           FORD             3000          4100
           JONES            2975          7075
           SCOTT            3000          8975
           SMITH             800          6775

        30 ALLEN            1600          1600
           BLAKE            2850          4450
           JAMES             950          5400
           MARTIN           1250          5050
           TURNER           1500          3700
           WARD             1250          4000

14 rows selected.
```

Нас интересует эта часть запроса:

```
  2   sum(sal) over
  3     (partition by deptno
```

```
4        order by ename
5      rows 2 preceding) sliding_total
```

Конструкция, определяющая фрагментацию, приводит к вычислению **SUM(SAL)** по отделам, независимо от других групп (значение **SUM(SAL)** сбрасывается при изменении номера отдела). Конструкция **ORDER BY ENAME** приводит к сортировке данных в каждом отделе по столбцу **ENAME**; это позволяет с помощью конструкции окна, rows 2 preceding, при суммировании зарплат обращаться к двум предыдущим строкам в соответствии с заданным порядком сортировки. Например, значение в столбце **SLIDING_TOTAL** для сотрудника **SMITH — 6775**, что равно сумме значений 800, 3000 и **2975**. Это сумма зарплат в строке для **SMITH** и двух предыдущих строках окна.

Можно создавать окна по двум критериям: по диапазону (RANGE) значений данных или по смещению (**ROWS**) относительно текущей строки. Конструкция RANGE уже встречалась ранее, **RANGE UNBOUNDED PRECEDING** например. Она требует брать все строки вплоть до текущей, в соответствии с порядком, задаваемым конструкцией ORDER BY. Следует помнить, что для использования окон необходимо задавать конструкцию ORDER BY. Сейчас мы рассмотрим задание окон с помощью конструкций ROWS и RANGE, а затем другие способы задания окон.

Окнадиапазона

Окна диапазона объединяют строки в соответствии с заданным порядком. Если в запросе сказано, например, "range 5 preceding", то будет сгенерировано перемещающееся окно, включающее предыдущие строки группы, отстоящие от текущей строки не более чем на 5 строк. Диапазон можно задавать в виде числового выражения или выражения, значением которого является дата. Применять конструкцию **RANGE** с другими типами данных нельзя.

Если имеется таблица **EMP** со столбцом **HIREDATE** типа даты и задана аналитическая функция

```
count(*) over (order by hiredate asc range 100 preceding)
```

она найдет все предыдущие строки фрагмента, значение которых в столбце **HIREDATE** лежит в пределах 100 дней от значения HIREDATE текущей строки. В этом случае, поскольку данные сортируются по возрастанию (ASC), значения в окне будут включать все строки текущей группы, у которых значение в столбце HIREDATE меньше значения HIREDATE текущей строки, но не более чем на 100 дней. Если использовать функцию

```
count(*) over (order by hiredate desc range 100 preceding)
```

и сортировать фрагмент по убыванию (**DESC**), базовая логика работы останется той же, но, поскольку группа отсортирована иначе, в окно попадет другой набор строк. В рассматриваемом случае функция найдет все строки, предшествующие текущей, где значение в поле **HIREDATE** больше значения **HIREDATE** в текущей строке, но не более чем на 100 дней. Пример поможет это прояснить. Я буду использовать запрос с аналитической функцией **FIRST_VALUE**. Эта функция возвращает вычисленное значение для первой строки окна. Так мы легко сможем понять, где начинается окно:

```
scott@TKYTE816>-select ename, sal, hiredate, hiredate-100 windowtop,
  2    first_value(ename)
  3    over (order by hiredate asc
  4         range 100 preceding) ename_prec,
  5    first_value(hiredate)
  6    over (order by hiredate asc
  7         range 100 preceding) hiredate_prec
  8  from emp
  9  order by hiredate asc
 10 /
```

ENAME	SAL	HIREDATE	WINDOW_TOP	ENAME_PREC	HIREDATE_
SMITH	800	17-DEC-80	08-SEP-80	SMITH	17-DEC-80
ALLEN	1600	20-FEB-81	12-NOV-80	SMITH	17-DEC-80
WARD	1250	22-FEB-81	14-NOV-80	SMITH	17-DEC-80
JONES	2975	**02-APR-81**	23-DEC-80	ALLEN	20-FEB-81
BLAKE	2850	01-MAY-81	21-JAN-81	ALLEN	20-FEB-81
CLARK	2450	09-JUN-81	01-MAR-81	JONES	**02-APR-81**
TURNER	1500	08-SEP-81	31-MAY-81	CLARK	09-JUN-81
MARTIN	1250	28-SEP-81	20-JUN-81	TURNER	08-SEP-81
KING	5000	17-NOV-81	09-AUG-81	TURNER	08-SEP-81
FORD	3000	03-DEC-81	25-AUG-81	TURNER	08-SEP-81
JAMES	950	03-DEC-81	25-AUG-81	TURNER	08-SEP-81
MILLER	1300	23-JAN-82	15-OCT-81	KING	17-NOV-81
SCOTT	3000	09-DEC-82	31-AUG-82	SCOTT	09-DEC-82
ADAMS	1100	12-JAN-83	04-OCT-82	SCOTT	09-DEC-82

14 rows selected.

Мы упорядочили один фрагмент по критерию **HIREDATE ASC.** При этом использовалась аналитическая функция **FIRST_VALUE** для поиска первого значения **ENAME** и первого значения **HIREDATE** в соответствующем окне. Посмотрев на строку данных для сотрудника **CLARK,** можно обнаружить, что для него значение в столбце **HIREDATE** — 9 июня 1981 года, **09-JUN-81,** а дата за 100 дней до этой соответствует 1 марта 1981 года, **01-MAR-81.** Для удобства эта дата помещена в столбец **WINDOWTOP.** Аналитическая функция затем вычисляется для всех строк отсортированного фрагмента, предшествующих строке для сотрудника **CLARK** и имеющих значение в столбце **HIREDATE** в диапазоне с **01-MAR-81** по **09-JUN-81.** Первое значение **ENAME** для этого окна — **JONES.** Это имя и выдает аналитическая функция в столбце **ENAME_PREC.**

Упорядочив данные по критерию **HIREDATE DESC,** мы получим:

```
scott@TKYTE816> select ename, sal, hiredate, hiredate+100 windowtop,
  2    first_value(ename)
  3    over (order by hiredate desc
  4         range 100 preceding) ename_prec,
  5    first_value(hiredate)
  6    over (order by hiredate desc
  7         range 100 preceding) hiredate_prec
  8   from emp
  9   order by hiredate desc
 10 /
```

```
ENAME              SAL HIREDATE  WINDOWTOP ENAME_PREC HIREDATE_

ADAMS             1100 12-JAN-83 22-APR-83 ADAMS      12-JAN-83
SCOTT             3000 09-DEC-82 19-MAR-83 ADAMS      12-JAN-83
MILLER            1300 23-JAN-82 03-MAY-82 MILLER     23-JAN-82
FORD              3000 03-DEC-81 13-MAR-82 MILLER     23-JAN-82
JAMES              950 03-DEC-81 13-MAR-82 MILLER     23-JAN-82
KING              5000 17-NOV-81 25-FEB-82 MILLER     23-JAN-82
MARTIN            1250 28-SEP-81 06-JAN-82 FORD       03-DEC-81
TURNER            1500 08-SEP-81 17-DEC-81 FORD       03-DEC-81
CLARK             2450 09-JUN-81 17-SEP-81 TURNER     08-SEP-81
BLAKE             2850 01-MAY-81 09-AUG-81 CLARK      09-JUN-81
JONES             2975 02-APR-81 11-JUL-81 CLARK      09-JUN-81
WARD              1250 22-FEB-81 02-JUN-81 BLAKE      01-MAY-81
ALLEN             1600 20-FEB-81 31-MAY-81 BLAKE      01-MAY-81
SMITH              800 17-DEC-80 27-MAR-81 WARD       22-FEB-81

14 rows selected.
```

Если снова обратиться к строке сотрудника **CLARK**, окажется, что выбрано другое окно, поскольку данные фрагмента отсортированы по-иному. Окно для строки **CLARK** по условию **RANGE 100 PRECEDING** теперь доходит до строки **TURNER**, поскольку значение **HIREDATE** для **TURNER** — последняя дата среди значений **HIREDATE** в строках, предшествующих строке **CLARK**, отличающаяся не более чем на 100 дней.

Иногда достаточно сложно понять, какие значения будут входить в диапазон. Я считаю использование функции **FIRST_VALUE** удобным методом, помогающим увидеть диапазоны окна и проверить, корректно ли установлены параметры. Теперь, представив диапазоны окон, мы используем их для вычисления чего-то более существенного. Пусть необходимо выбрать зарплату каждого сотрудника и среднюю зарплату всех принятых на работу в течение 100 предыдущих дней, а также среднюю зарплату всех принятых на работу в течение 100 следующих дней. Соответствующий запрос будет выглядеть так:

```
scott@TKYTE816> select ename, hiredate, sal,
  2    avg(sal)
  3    over (order by hiredate asc range 100 preceding)
  4      avg_sal_100_days_before,
  5    avg(sal)
  6    over (order by hiredate desc range 100 preceding)
  7      avg_sal_100_days_after
  8  from emp
  9  order by
  8  /

ENAME    HIREDATE        SAL  AVG_SAL_100_DAYS_BEPORE  AVG_SAL_100_DAYS_AFTER

SMITH    17-DEC-80    800.00                800.00                 1216.67
ALLEN    20-FEB-81   1600.00               1200.00                 2168.75
WARD     22-FEB-81   1250.00               1216.67                 2358.33
JONES    02-APR-81   2975.00               1941.67                 2758.33
BLAKE    01-MAY-81   2850.00               2168.75                 2650.00
CLARK    09-JUN-81   2450.00               2758.33                 1975.00
TURNER   08-SEP-81   1500.00               1975.00                 2340.00
```

```
MARTIN  28-SEP-81  1250.00          1375.00          2550.00
KING    17-NOV-81  5000.00          2583.33          2562.50
JAMES   03-DEC-81   950.00          2340.00          1750.00
FORD    03-DEC-81  3000.00          2340.00          1750.00
MILLER  23-JAN-82  1300.00          2562.50          1300.00
SCOTT   09-DEC-82  3000.00          3000.00          2050.00
ADAMS   12-JAN-83  1100.00          2050.00          1100.00

14 rows selected.
```

Если теперь снова обратиться к строке для сотрудника **CLARK,** то, поскольку мы уже понимаем, какое окно в группе будет с ней связано, легко убедиться, что средняя зарплата **(2758,33)** равна **(2450+2850+2975)/3.** Это средняя зарплата для строки **CLARK** и строк, предшествующих CLARK (это строки для сотрудников **JONES** и **BLAKE)** при упорядочении данных по возрастанию. С другой стороны, средняя зарплата **1975,00** равна **(2450+1500)/2.** Это средняя зарплата для строки **CLARK** и строк, предшествующих CLARK при упорядочении данных по убыванию. С помощью этого запроса можно одновременно вычислить среднюю зарплату для сотрудников, принятых на работу за 100 дней до и за 100 дней после сотрудника **CLARK.**

Окна диапазона можно задавать только по данным типа **NUMBER** или **DATE,** поскольку нельзя добавить или вычесть N единиц из значения типа **VARCHAR2.** Еще одно ограничение для таких окон состоит в том, что в конструкции **ORDER BY** может быть только один столбец — диапазоны по природе своей одномерны. Нельзя задать диапазон в N-мерном пространстве.

Окна строк

Окна срок задаются в физических единицах, строках. Перепишем вступительный пример из предыдущего раздела, задав окно строк:

```
count (*) over (order by x ROWS 5 preceding)
```

Это окно будет включать до 6 строк: текущую и пять предыдущих (порядок определяется конструкцией **ORDER BY).** Для окон по строкам нет ограничений, присущих окнам по диапазону; данные могут быть любого типа и упорядочивать можно по любому количеству столбцов. Вот пример, сходный с рассмотренным ранее:

```
scott@TKYTE816> select ename, sal, hiredate,
  2    first_value(ename)
  3    over (order by hiredate asc
  4         rows 5 preceding) ename_prec,
  5    first_value(hiredate)
  6    over (order by hiredate asc
  7         rows 5 preceding) hiredate_prec
  8  from emp
  9  order by hiredate asc
 10  /

ENAME          SAL HIREDATE  ENAME_PREC HIREDATE_

SMITH        800.00 17-DEC-80 SMITH      17-DEC-80
ALLEN       1600.00 20-FEB-81 SMITH      17-DEC-80
WARD        1250.00 22-EEB-81 SMITH      17-DEC-80
```

```
JONES        2975.00 02-APR-81 SMITH       17-DEC-80
BLAKE        2850.00 01-MAY-81 SMITH       17-DEC-80
CLARK        2450.00 09-JUN-81 SMITH       17-DEC-80
TURNER       1500.00 08-SEP-81 ALLEN       20-FEB-81
MARTIN       1250.00 28-SEP-81 WARD        22-FEB-81
KING         5000.00 17-NOV-81 JONES       02-APR-81
JAMES         950.00 03-DEC-81 BLAKE       01-MAY-81
FORD         3000.00 03-DEC-81 CLARK       09-JUN-81
MILLER       1300.00 23-JAN-82 TURNER      08-SEP-81
SCOTT        3000.00 09-DEC-82 MARTIN      28-SEP-81
ADAMS        1100.00 12-JAN-83 KING        17-NOV-81
14 rows selected.
```

Взглянув на строку для сотрудника **CLARK**, можно увидеть, что первой в окне **ROWS 5 PRECEDING** следует строка для сотрудника **SMITH** — она просто пятая перед строкой для **CLARK** в соответствии с заданным порядком. Строка для сотрудника **SMITH** будет первой в окне и для всех предыдущих строк (для **BLAKE**, **JONES** и т.д.). Дело в том, что строка для **SMITH** — первая в данной группе (она будет первой и для самой себя). При сортировке группы по возрастанию окна изменяются:

```
scott@TKYTE816> select ename, sal, hiredate,
  2    first_value(ename)
  3    over (order by hiredate desc
  4          rows 5 preceding) ename_prec,
  5    first_value(hiredate)
  6    over (order by hiredate desc
  7          rows 5 preceding) hiredate_prec
  8  from emp
  9  order by hiredate desc
 10 /

ENAME            SAL HIREDATE  ENAME_PREC HIREDATE_

ADAMS        1100.00 12-JAN-83 ADAMS       12-JAN-83
SCOTT        3000.00 09-DEC-82 ADAMS       12-JAN-83
MILLER       1300.00 23-JAN-82 ADAMS       12-JAN-83
JAMES         950.00 03-DEC-81 ADAMS       12-JAN-83
FORD         3000.00 03-DEC-81 ADAMS       12-JAN-83
KING         5000.00 17-NOV-81 ADAMS       12-JAN-83
MARTIN       1250.00 28-SEP-81 SCOTT       09-DEC-82
TURNER       1500.00 08-SEP-81 MILLER      23-JAN-82
CLARK        2450.00 09-JUN-81 JAMES       03-DEC-81
BLAKE        2850.00 01-MAY-81 FORD        03-DEC-81
JONES        2975.00 02-APR-81 KING        17-NOV-81
WARD         1250.00 22-FEB-81 MARTIN      28-SEP-81
ALLEN        1600.00 20-FEB-81 TURNER      08-SEP-81
SMITH         800.00 17-DEC-80 CLARK       09-JUN-81

14 rows selected.
```

Теперь первое значение для набора из 5 строк, предшествующих в группе строке для сотрудника **CLARK**, — строка для сотрудника **JAMES**. Теперь можно вычислить сред-

нюю зарплату для указанного сотрудника и пяти принятых на работу до него и после него:

```
scott@TKYTE816> select ename, hiredate, sal,
  2    avg(sal)
  3    over (order by hiredate asc rows 5 preceding) avg_5_before,
  4      count(*)
  5    over (order by hiredate asc rows 5 preceding) obs_before,
  6      avg(sal)
  7   over (order by hiredate desc rows 5 preceding) avg_5_after,
  8      count(*)
  9   over (order by hiredate desc rows 5 preceding) obs_after
 10  from emp
 11  order by hiredate
 12  /
```

ENAME	HIREDATE	SAL	AVG_5_BEFORE	OBS_BEPOPE	AVG_5_APTER	OBS_AFTER
SMITH	17-DEC-80	800.00	800.00	1.00	1987.50	6.00
ALLEN	20-FEB-81	1600.00	1200.00	2.00	2104.17	6.00
WARD	22-FEB-81	1250.00	1216.67	3.00	2045.83	6.00
JONES	02-APR-81	2975.00	1656.25	4.00	2670.83	6.00
BLAKE	01-MAY-81	2850.00	1895.00	5.00	2675.00	6.00
CLARK	09-JUN-81	2450.00	1987.50	6.00	2358.33	6.00
TURNER	08-SEP-81	1500.00	2104.17	6.00	2166.67	6.00
MARTIN	28-SEP-81	1250.00	2045.83	6.00	2416.67	6.00
KING	17-NOV-81	5000.00	2670.83	6.00	2391.67	6.00
JAMES	03-DEC-81	950.00	2333.33	6.00	1587.50	4.00
FORD	03-DEC-81	3000.00	2358.33	6.00	1870.00	5.00
MILLER	23-JAN-82	1300.00	2166.67	6.00	1800.00	3.00
SCOTT	09-DEC-82	3000.00	2416.67	6.00	2050.00	2.00
ADAMS	12-JAN-83	1100.00	2391.67	6.00	1100.00	1.00

```
14 rows selected.
```

Обратите внимание, что я выбирал также значение **COUNT(*)**. Это позволяет понять, по какому количеству строк было вычислено среднее значение. Можно явно увидеть, что для вычисления средней зарплаты сотрудников, принятых до сотрудника **ALLEN**, использовалось только 2 записи, а для вычисления средней зарплаты сотрудников, нанятых после него, — 6. В том месте группы, где находится строка для сотрудника **ALLEN,** есть только 1 предыдущая запись, а при вычислении аналитической функции используются все имеющиеся строки.

Заданиеокон

Теперь, понимая различие между окнами диапазонов и окнами строк, можно изучать способы задания окон.

В простейшем случае, окно задается с помощью одной из трех следующих взаимоисключающих конструкций.

- **UNBOUNDED PRECEDING.** Окно начинается с первой строки текущей группы и заканчивается текущей обрабатываемой строкой.

- **CURRENT ROW.** Окно начинается (и заканчивается) текущей строкой.
- **Числовое_выражение PRECEDING.** Окно начинается со строки за **числовое_выражение** строк до текущей, если оно задается по строкам, или со строки, меньшей по значению столбца, упомянутого в конструкции **ORDER BY,** не более чем на **числовое выражение,** если оно задается по диапазону.

Окно **CURRENT ROW** в простейшем виде, вероятно, никогда не используется, поскольку ограничивает применение аналитической функции одной текущей строкой, а для этого аналитические функции не нужны. В более сложном случае для окна задается также конструкция **BETWEEN. В** ней **CURRENT ROW** можно указывать в качестве начальной или конечной строки окна. Начальную и конечную строку окна в конструкции **BETWEEN** можно задавать с использованием любой из перечисленных выше конструкций и еще одной, дополнительной:

- **Числовое_выражение FOLLOWING.** Окно заканчивается (или начинается) со строки, через **числовое_выражение** строк после текущей, если оно задается по строкам, или со строки, большей по значению столбца, упомянутого в конструкции **ORDER BY,** не более чем на **числовое_выражение,** если оно задается по диапазону.

Рассмотрим ряд примеров такого задания окон:

```
scott@TKYTE816> select deptno, ename, hiredate,
  2    count(*) over  (partition by deptno
  3                    order by hiredate nulls first
  4                   range 100 preceding)  cnt_range,
  5    count(*) over  (partition by deptno
  6                    order by hiredate nulls first
  7                   rows 2 preceding)  cnt_rows
  8  from emp
  9  where deptno in (10, 20)
 10  order by deptno, hiredate
 11  /
```

DEPTNO	ENAME	HIREDATE	CNT_RANGE	CNT_ROWS
10	CLARK	09-JUN-81	1	1
	KING	17-NOV-81	1	2
	MILLER	23-JAN-82	2	3
20	SMITH	17-DEC-80	1	1
	JONES	02-APR-81	1	2
	FORD	03-DEC-81	1	3
	SCOTT	09-DEC-82	1	3
	ADAMS	12-JAN-83	2	3

```
8 rows selected.
```

Как видите, окно **RANGE 100 PRECEDING** содержит только строки текущего фрагмента, предшествующие текущей строке, и те, значение которых **HIREDATE** находится в диапазоне **HIREDATE-100** и **HIREDATE** относительно текущей. В данном случае таких строк всегда 1 или 2, т.е. интервал между приемом людей на работу обычно пре-

вышает 100 дней (за исключением двух случаев). Окно **ROWS 2 PRECEDING**, однако, содержит от 1 до 3 строк (это определяется тем, как далеко текущая строка находится от начала группы). Для первой строки группы имеем значение 1 (предыдущих строк нет). Для следующей строки в группе таких строк 2. Наконец, для третьей и далее строк значение **COUNT(*)** остается постоянным, поскольку мы считаем только текущую строку и две предыдущие.

Теперь рассмотрим использование конструкции **BETWEEN**. Все заданные до сих пор окна заканчивались текущей строкой и возвращались по результирующему множеству в поисках дополнительной информации. Можно задать окно так, что обрабатываемая строка не будет последней, а окажется где-то в середине окна. Например:

```
scott@TKYTE816> select ename, hiredate,
  2    first_value(ename) over
  3       (order by hiredate asc
  4       range between 100 preceding and 100 following),
  5       last_value(ename) over
  6       (order by hiredate asc
  7       range between 100 preceding and 100 following)
  8  from emp
  9  order by hiredate asc
 10  /
```

ENAME	HIREDATE	FIRST_VALU	LAST_VALUE
SMITH	17-DEC-80	SMITH	WARD
ALLEN	20-FEB-81	SMITH	BLAKE
WARD	22-FEB-81	SMITH	BLAKE
JONES	02-APR-81	ALLEN	CLARK
BLAKE	01-MAY-81	ALLEN	CLARK
CLARK	09-JUN-81	JONES	**TURNER**
TURNER	08-SEP-81	CLARK	JAMES
MARTIN	28-SEP-81	TURNER	JAMES
KING	17-NOV-81	TURNER	MILLER
FORD	03-DEC-81	TURNER	MILLER
JAMES	03-DEC-81	TURNER	MILLER
MILLER	23-JAN-82	KING	MILLER
SCOTT	09-DEC-82	SCOTT	ADAMS
ADAMS	12-JAN-83	SCOTT	ADAMS

```
14 rows selected.
```

Обратившись снова к строке для сотрудника **CLARK**, можно убедиться, что окно начинается со строки для **JONES** и продолжается до строки для сотрудника **TURNER**. Теперь в окно входят строки для тех, кто принят на работу за 100 дней до и (а не или, как прежде) после текущего сотрудника.

Итак, теперь мы хорошо знаем синтаксис четырех компонентов вызова аналитической функции. Это:

- имя функции;
- конструкция фрагментации, используемая для разбиения результирующего множества на независимые группы;

- **конструкция ORDER BY,** сортирующая данные в группе **для оконных** функций;
- конструкция окна, задающая набор строк, к которым применяется аналитическая функция.

ИМЯ_ФУНКЦИИ(<аргумент>, <аргумент>, ...)
OVER
(конструкция фрагментации> <конструкция упорядочений <конструкция окна>)

Рассмотрим кратко предлагаемые функции.

Функции

Сервер предлагает более 26 аналитических функций. Имена некоторых из них совпадают с именами функций агрегирования, например **AVG** и **SUM.** Другие, с новыми именами, обеспечивают новые возможности. В этом разделе будут перечислены имеющиеся функции и кратко описано их назначение.

Аналитическая функция	*Назначение*
AVG([DISTINCT | ALL] выражение)	Используется для вычисления среднего значения выражения в пределах группы и окна. Для поиска среднего после удаления дублирующихся значений можно указывать ключевое слово **DISTINCT,**
CORR (выражение, выражение)	Выдает коэффициент корреляции для пары выражений, возвращающих числовые значения. Это сокращение для выражения: COVAR_POP<выражение1, выражение2) / STDDEV_POP(выражение!.) * STDDEV_POP(выражение2)). В статистическом смысле, *корреляция* — это степень связи между переменными. Связь между переменными означает, что значение одной переменной можно в определенной степени предсказать по значению другой. Коэффициент корреляции представляет степень корреляции в виде числа в диапазоне от -1 (высокая обратная корреляция) до 1 (высокая корреляция). Значение 0 соответствует отсутствию корреляции,
COUNT([DISTINCT] [*] [выражение])	Эта функция считает строки в группах. Если указать * или любую константу, кроме **NULL,** функция **count** будет считать все строки. Если указать **выражение,** функция **count** будет считать строки, для которых выражение имеет значение не **NULL.** Можно задавать модификатор **DISTINCT,** чтобы считать строки в группах после удаления дублирующихся строк.
COVAR_POP(выражение, выражение)	Возвращает *ковариацию генеральной совокупности* (population covariance) пары выражений с числовыми значениями.
COVAR_SAMP (выражение, выражение)	Возвращает *выборочную ковариацию* (sample covariance) пары выражений с числовыми значениями.

Аналитическая функция	*Назначение*
CUME_DIST	Вычисляет относительную позицию строки в группе. Функция **CUME_DIST** всегда возвращает число большее О и меньше или равное 1. Это число представляет "позицию" строки в группе из N арок. В группе из трех строк, например, возвращаются следующие значения кумулятивного распределения: 1/3, 2/3 и 3/3.
DENSE_RANK	Эта функция вычисляет относительный ранг каждой возвращаемой запросом строки по отношению к другим строкам, основываясь на значениях выражений в конструкции **ORDER** BY. Данные в группе сортируются в соответствии с конструкцией **ORDER BY,** а затем каждой строке поочередно присваивается числовой ранг, начиная с 1. Ранг увеличивается при каждом изменении значений выражений, входящих в конструкцию ORDER BY. Строки с одинаковыми значениями получают один и тот же ранг (при этом сравнении значения NULL считаются одинаковыми). Возвращаемый этой функцией "плотный" ранг дает ранговые значения без промежутков. Сравните с представленной далее функцией **RANK.**
FIRST_VALUE	Возвращает первое значение в группе.
LAG(выражение, <смещение>, <стандартное значение>)	Функция **LAG** дает доступ к другим строкам результирующего множества, избавляя от необходимости выполнять самосоединения. Она позволяет работать с курсором как с массивом. Можно ссылаться на строки, предшествующие текущей строке в группе. О том, как обращаться к следующим строкам в группе, см. в описании функции **LEAD.** Смещение - это положительное целое число со стандартным значением 1 (предыдущая строка). Стандартное значение возвращается, если индекс выходит за пределы окна (для первой строки группы будет возвращено стандартное значение).
LAST_VALUE	Возвращает последнее значение в группе.
LEAD(выражение, <смещение>, <стандартное значение>)	Функция **LEAD** противоположна функции **LAG.** Если функция **LAG** дает доступ к предшествующим строкам группы, то функция **LEAD** позволяет обращаться к строкам, следующим за текущей. Смещение — это положительное целое число со стандартным значением 1 (следующая строка). Стандартное значение возвращается, если индекс выходит за пределы окна (для последней строки группы будет возвращено стандартное значение).
MAX(выражение)	Находит максимальное значение выражения в пределах окна в группе.
MIN(выражение)	Находит минимальное значение выражения в пределах окна в группе.

Аналитическая функция	*Назначение*
NTILE(выражение)	Делит группу на фрагменты по значению выражения. Например, если выражение = 4, то каждой строке в группе присваивается число от 1 до 4 в соответствии с фрагментом, з которую она попадает. Если в группе 20 строк, первые 5 получат значение 1, следующие 5 — значение 2 и т.д. Если количество строк в группе не делится на значение выражения без остатка, строки распределяются так, что ни в одном фрагменте количество строк не превосходит минимальное количество в других фрагментах более чем на 1, причем дополнительные строки будут в группах с меньшими номера фрагмента. Например, если снова выражение = 4, а количество строк = 21, в первом фрагменте будет 6 строк, во втором и последующих - 5.
PERCENT RANK	Аналогична функции **CUME_DIST** (кумулятивное распределение). Вычисляет ранг строки в группе минус 1, деленный на количество обрабатываемых строк минус 1. Эта функция всегда возвращает значения в диапазоне от 0 до 1 включительно.
RANK	Эта функция вычисляет относительный ранг каждой строки, возвращаемой запросом, на основе значений выражений, входящих в конструкцию **ORDER BY.** Данные в группе сортируются в соответствии с конструкцией **ORDER BY,** а затем каждой строке поочередно присваивается числовой ранг, начиная с 1. Строки с одинаковыми значениями выражений, входящих в конструкцию **ORDER BY,** получают одинаковый ранг, но если две строки получат одинаковый ранг, следующее значение ранга пропускается. Если две строки получили ранг 1, строки с рангом 2 не будет; следующая строка в группе получит ранг 3. В этом отличие от функции **DENSE_RANK,** которая не пропускает значений.
RATIO_TO_REPORT (выражение)	Эта функция вычисляет значение **выражение** / **(sum(выражение))** по строкам группы. Это дает процент, который составляет значение текущей строки по отношению к **sum(выражение)**
REGR_xxxxxxx (выражение, выражение)	Эти функции линейной регрессии применяют стандартную линейную регрессию по методу наименьших квадратов к паре выражений. Предлагается 9 различных функций регрессии.
ROW NUMBER	Возвращает смещение строки по отношению к началу упорядоченной группы. Может использоваться для последовательной нумерации строк, упорядоченных по определенным критериям.
STDDEV(выражение)	Вычисляет *стандартное (среднеквадратичное) отклонение* (standard deviation) текущей строки по отношению к группе.

Аналитическая функция	*Назначение*
STDDEV_POP (выражение)	Эта функция вычисляет *стандартное отклонение генеральной совокупности* (population standard deviation) и возвращает квадратный корень из *дисперсии генеральной совокупности* (population variance). Она возвращает значение, совпадающее с квадратным корнем из результата функции VAR_POP.
STDDEV_SAMP (выражение)	Эта функция вычисляет *накопленное стандартное отклонение выборки* (cumulative sample standard deviation) и возвращает квадратный корень *выборочной дисперсии* (sample variance). Она возвращает значение, совпадающее с квадратным корнем из результата функции VAR_SAMP.
SUM(выражение)	Вычисляет общую сумму значений выражения для группы.
VAR_TOP(выражение)	Эта функция возвращает дисперсию генеральной совокупности для набора числовых значений (значения NULL игнорируются). Функция VAR_POP вычисляет значение: `(SUM(выражение*выражение) – SUM(выражение)*SUM(выражение) / COUNT(выражение)) / COUNT(выражение)`
VAR_SAMP(выражение)	Эта функция возвращает выборочную дисперсию для набора числовых значений (значения NULL игнорируются). Она вычисляет значение: `(SUM(выражение*выражение) – SUM(выражение)*SUM(выражение) / COUNT(выражение)) / (COUNT(выражение) - 1)`
VARIANCE(выражение)	Возвращает дисперсию для выражения. Сервер Oracle вычисляет дисперсию как: • 0, если количество строк в группе = 1; • VAR_SAMP, если количество строк в группе > 1.

Примеры

Теперь можно переходить к самой интересной части — возможностям, предоставляемым аналитическими функциями. Приводимые примеры не демонстрируют все возможности, а лишь дают начальное представление.

Запрос первых N

Мне часто задают вопрос: "Как получить первых N записей набора полей?". До появления аналитических функций ответить на такой вопрос было очень трудно.

С запросами первых N записей, однако, бывают трудности связанные в основном с формулировкой задачи. Это надо учитывать при проектировании отчетов. Рассмотрим следующее, вполне разумное на первый взгляд требование: получить для каждого отдела трех наиболее высокооплачиваемых специалистов по продажам.

Однако эта задача неоднозначна из-за возможного повторения значений: в отделе может быть четыре человека с одинаково огромной зарплатой, и что тогда делать?

Я могу предложить как минимум три одинаково разумных интерпретации этого требования, причем каждой интерпретации может соответствовать и не три записи! Требование можно интерпретировать так.

- Выдать список специалистов по продажам, имеющих одну из трех максимальных зарплат. Другими словами, найти все различные значения зарплаты, отсортировать, выбрать три наибольших, и вернуть всех сотрудников, зарплата которых совпадает с одним из этих трех значений.
- Выдать до трех человек с максимальными зарплатами. Если четыре человека имеют одинаковую максимальную зарплату, в ответ не должно выдаваться ни одной строки. Если два сотрудника имеют максимальную зарплату и два — следующую по значению, ответ будет предполагать две строки (два сотрудника с максимальной зарплатой).
- Отсортировать специалистов по продажам по убыванию зарплат. Вернуть первые три строки. Если в отделе менее трех специалистов по продажам, в результате будет менее трех записей.

После дополнительных вопросов и уточнений оказывается, что некоторым необходима первая интерпретация; другим — вторая или третья. Давайте рассмотрим, как с помощью аналитических функций сформулировать все три запроса, и как это делалось без них.

Для этих примеров используем таблицу **SCOTT.EMP**. Сначала реализуем запрос "Выдать список специалистов по продажам в каждом отделе, имеющих одну из трех максимальных зарплат":

```
scott@TKYTE816> select *
  2    from (select deptno, ename, sal,
  3          dense_rank()
  4            over (partition by deptno
  5             order by sal desc)
  6          dr from emp)
  7   where dr <= 3
  8 order by deptno, sal desc
  9  /
```

DEPTNO	ENAME	SAL	DR
10	KING	5000	1
	CLARK	2450	2
	MILLER	1300	3
20	SCOTT	3000	1
	FORD	3000	1
	JONES	2975	2
	ADAMS	1100	3
30	BLAKE	2850	1
	ALLEN	1600	2
	TURNER	1500	3

```
10 rows selected.
```

Здесь для получения трех максимальных зарплат была использована функция DENSE_RANK(). Мы присвоили записям непрерывные ранговые значения по столбцу sal и отсортировали результат по убыванию. Если обратиться к описанию функций, оказывается, что при непрерывном ранжировании значения ранга не пропускаются и две строки с одинаковыми значениями получают одинаковый ранг. Поэтому после построения результирующего множества в виде подставляемого представления, можно просто выбирать все строки с "плотным" рангом не более трех. В результате для каждого отдела будут получены все сотрудники с одной из трех максимальных зарплат в отделе. Для сравнения выберем функцию RANK и сравним, что происходит при обнаружении дублирующихся значений:

```
scott@TKYTE816> select deptno, ename, sal,
  2         dense_rank()
  3           over (partition by deptno
  4                order by sal desc) dr,
  5         rank()
  6           over (partition by deptno
  7                order by sal desc) r
  8    from emp
  9   order by deptno, sal desc
 10  /
```

DEPTNO	ENAME	SAL	DR	R
10	KING	5000	1	1
	CLARK	2450	2	2
	MILLER	1300	3	3
20	SCOTT	3000	1	1
	FORD	3000	1	1
	JONES	2975	2	3
	ADAMS	1100	3	4
	SMITH	800	4	5
30	BLAKE	2850	1	1
	ALLEN	1600	2	2
	TURNER	1500	3	3
	WARD	1250	4	4
	MARTIN	1250	4	4
	JAMES	950	5	6

```
14 rows selected.
```

Если бы использовалась функция **RANK**, сотрудник **ADAMS** (получивший ранг 4) не вошел бы в результирующее множество, но он — один из сотрудников отдела 20, получивших одну из трех максимальных зарплат, так что в результат он попадать должен. В данном случае использование функции **RANK** вместо **DENSE_RANK** привело бы к неправильному ответу на поставленный вопрос.

Наконец, пришлось использовать подставляемое представление и задать псевдоним **DR** для результатов аналитической функции **dense_rank()**. Дело в том, что нельзя использовать аналитические функции в конструкциях **WHERE** или **HAVING** непосредственно, так что пришлось выбрать результат в представление, а затем отфильтровать,

оставив только необходимые строки. Использование подставляемого представления с условием — типичная конструкция для многих примеров в этой главе.

Теперь вернемся к запросу "Выдать не более трех человек с максимальными зарплатами по каждому отделу":

```
scott@TKYTE816> select *
  2  from (select deptno, ename, sal,
  3        count(*) over (partition by deptno
  4                       order by sal desc
  5                       range unbounded preceding)
  6        cnt from emp)
  7  where cnt <= 3
  8  order by deptno, sal desc
  9  /

    DEPTNO ENAME             SAL        CNT

        10 KING             5000          1
           CLARK            2450          2
           MILLER           1300          3

        20 SCOTT            3000          2
           FORD             3000          2
           JONES            2975          3

        30 BLAKE            2850          1
           ALLEN            1600          2
           TURNER           1500          3

9 rows selected.
```

Этот запрос немного нетривиален. Мы подсчитываем все записи в окне, предшествующие текущей, при сортировке по зарплате. Диапазон **RANGE UNBOUNDED PRECEDING** задает окно, включающее все записи, зарплата в которых больше или равна зарплате в текущей записи, поскольку сортировка выполнена по убыванию **(DESC).** Подсчитывая всех сотрудников с такой же или более высокой зарплатой, можно выбирать только строки, в которых значение этого количества (CNT), меньше или равно 3. Обратите внимание, что в отделе 20 для сотрудников **SCOTT** и **FORD** возвращается значение 2. Оба они получили наибольшую зарплату в отделе, так что попадают в окно друг для друга. Интересно отметить небольшое отличие, которое дает следующий запрос:

```
scott@TKYTE816> select *
  2  from (select deptno, ename, sal,
  3        count(*) over (partition by deptno
  4                       order by sal desc, ename
  5                       range unbounded preceding)
  6        cnt from emp)
  7  where cnt <= 3
  8  order by deptno, sal desc
  9  /

    DEPTNO ENAME             SAL        CNT

        10 KING             5000          1
```

```
        CLARK          2450          2
        MILLER         1300          3

     20 FORD           3000          1
        SCOTT          3000          2
        JONES          2975          3

     30 BLAKE          2850          1
        ALLEN          1600          2
        TURNER         1500          3

9 rows selected.
```

Обратите внимание, как добавление столбца в конструкцию **ORDER** BY повлияло на окно. Ранее сотрудники **FORD** и **SCOTT** оба имели в столбце **CNT** значение **2**. Причина в том, что окно строилось исключительно по столбцу зарплаты. Более избирательное окно дает другие результаты функции **COUNT**. Я привел этот пример, чтобы подчеркнуть, что функция окна зависит от обеих конструкций, **ORDER BY** и **RANGE**. Если фрагмент сортировался только по зарплате, строка для сотрудника **FORD** предшествовала строке для **SCOTT**, когда строка для **SCOTT** была текущей, а строка для **SCOTT**, в свою очередь, предшествовала строке для **FORD,** когда та была текущей. Только при сортировке по столбцам **SAL** и **ENAME** можно однозначно упорядочить строки для сотрудников **SCOTT** и **FORD** по отношению друг к другу.

Чтобы убедиться, что этот подход, предусматривающий использование функции **COUNT**, позволяет возвращать не более трех записей, давайте изменим данные так, чтобы максимальная зарплата была у большего числа сотрудников отдела:

```
scott@TKYTE816> update emp set sal = 99 where deptno = 30;

6 rows updated.

scott@TKYTE816> select *
  2  from (select deptno, ename, sal,
  3        count(*) over (partition by deptno
  4                       order by sal desc
  5                       range unbounded preceding)
  6         cnt from emp)
  7  where cnt <= 3
  8  order by deptno, sal desc
  9  /

    DEPTNO ENAME             SAL        CNT

        10 KING             5000          1
           CLARK            2450          2
           MILLER           1300          3

        20 SCOTT            3000          2
           FORD             3000          2
           JONES            2975          3

6 rows selected.
```

Теперь строк для отдела 30 в отчете нет, поскольку 6 сотрудников этого отдела имеют одинаковую зарплату. В поле **CNT** для всех них находится значение 6, которое никак не меньше или равно 3.

Перейдем теперь к последнему запросу: "Отсортировать специалистов по продажам по убыванию зарплат и вернуть первые три строки". Это легко сделать с помощью функции **ROW_NUMBER():**

```
scott@TKYTE816> select *
  2  from (select deptno, ename, sal,
  3        row_number() over (partition by deptno
  4                           order by sal desc)
  5        rn from emp)
  6  where rn <= 3
  7  /
```

DEPTNO	ENAME	SAL	RN
10	KING	5000	1
	CLARK	2450	2
	MILLER	1300	3
20	SCOTT	3000	1
	FORD	3000	2
	JONES	2975	3
30	ALLEN	99	1
	BLAKE	99	2
	MARTIN	99	3

```
9 rows selected.
```

При выполнении запроса каждый фрагмент сортируется по убыванию значений зарплат, после чего по мере обработки каждой строке фрагмента присваивается последовательный номер. После этого с помощью конструкции **WHERE** мы получаем только первые три строки каждого фрагмента. В примере с транспонированием результирующего множества мы используем такой же прием для преобразования строк в столбцы. Следует отметить, однако, что для отдела **DEPTNO=30** возвращаются в определенном смысле случайные строки. Если помните, информация в отделе 30 была изменена так, что все 6 сотрудников получили значение 99 в столбце зарплаты. Можно в некоторой степени управлять тем, какие три записи будут возвращаться, с помощью конструкции **ORDER BY**. Например, можно использовать конструкцию **ORDER BY SAL DESC, ENAME** для получения упорядоченной по фамилии информации о наиболее высокооплачиваемых сотрудниках, если несколько из них имеют одинаковую зарплату.

Интересно отметить, что с помощью функции **ROW_NUMBER** можно получать произвольную секцию данных из группы строк. Это может пригодиться в среде, не поддерживающей информацию о состоянии, когда надо выдавать данные постранично. Например, если необходимо выдавать данные из таблицы **EMP**, отсортированные по столбцу **ENAME**, группами по пять строк, можно использовать запрос следующего вида:

```
scott@TKYTE816> select ename, hiredate, sal
  2  from (select ename, hiredate, sal,
  3        row_number() over (order by ename)
  4         rn from emp)
  5  where rn between 5 and 10
  6  order by rn
  7  /
```

```
ENAME      HIREDATE          SAL

FORD       03-DEC-81        3000
JAMES      03-DEC-81         950
JONES      02-APR-81        2975
KING       17-NOV-81        5000
MARTIN     28-SEP-81        1250
MILLER     23-JAN-82        1300

6 rows selected.
```

И напоследок, чтобы продемонстрировать всю мощь аналитических функций, сравним запросы с аналитическими функциями с такими же запросами, где эти функции не используются. Для сравнения я создал таблицу T, являющуюся увеличенной во всех смыслах разновидностью таблицы **EMP**:

```
scott@TKYTE816> create table t
  2  as
  3  select object_name ename,
  4         mod(object_id,50) deptno,
  5          object_id sal
  6   from all_objects
  7  where rownum <= 1000
  8  /

Table created.

scott@TKYTE816> create index t_idx on t(deptno,sal desc);

Index created.

scotteTKYTE816> analyze table t
  2  compute statistics
  3  for table
  4  for all indexed columns
  5  for all indexes
  6  /

Table analyzed.
```

Мы создали индекс по этой таблице, позволяющий ответить на запросы, которые мы будем к ней выполнять. Теперь сравним тексты и производительность запросов с аналитическими функциями и без них. Для сравнения производительности я использовал установки **SQL_TRACE**, **TIMED_STATISTICS** и утилиту **TKPROF**. Подробнее об этих средствах и интерпретации результатов см. в главе 10, посвященной стратегиям и средствам настройки производительности:

```
scott@TKYTE816> select *
  2  from (select deptno, ename, sal,
  3        dense_rank() over (partition by deptno
  4                           order by sal desc)
  5        dr from t)
  6  where dr <= 3
  7  order by deptno, sal desc
  8  /
```

```
call     count       cpu    elapsed       disk      query    current        rows

Parse        1      0.00       0.00          0          0          0           0
Execute      2      0.00       0.00          0          0          0           0
Fetch       11      0.01       0.07          7         10         17         150

total       14      0.01       0.07          7         10         17         150

Misses in library cache during parse: 0
Optimizer goal: CHOOSE
Parsing user id: 54

Rows     Row Source Operation

    150  VIEW
    364   WINDOW SORT PUSHED RANK
   1000    TABLE ACCESS FULL T
**********************************

scott@TKYTE816> select deptno, ename, sal
  2  from t el
  3  where sal in (select sal
  4                from (select distinct sal , deptno
  5                      from t e3
  6                     order by deptno, sal desc) e2
  7                where e2.deptno = e1.deptno
  8                and rownum <= 3)
  9   order by deptno, sal desc
 10  /

call     count       cpu    elapsed       disk      query    current        rows

Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch       11      0.80       0.80          0      10010      12012         150

total       13      0.80       0.80          0      10010      12012         150

Misses in library cache during parse: 0
Optimizer goal: CHOOSE
Parsing user id: 54

Rows     Row Source Operation

    150  SORT ORDER BY
    150   FILTER
   1001    TABLE ACCESS FULL T
   1000    FILTER
   3700     COUNT STOPKEY
   2850      VIEW
   2850       SORT ORDER BY STOPKEY
  20654        SORT UNIQUE
  20654         TABLE ACCESS FULL T
```

Оба представленных выше запроса возвращают данные о трех сотрудниках, которые получают наибольшие зарплаты. Запрос, использующий аналитические функции, позволяет получить эти сведения без особых усилий: для этого требуется 0,01 секунды процессорного времени и 27 логических операций ввода-вывода. А реляционный запрос требует выполнения более 22000 логических операций ввода-вывода, и на это уходит 0,80 секунды процессорного времени. Для выполнения запроса, не использующего аналитические функции, необходимо выполнять подзапрос для каждой строки в таблице T, чтобы найти три наибольших зарплаты в данном отделе. Этот запрос не только медленнее выполняется, но и сложен для написания. Его производительность можно повысить с помощью подсказок, но при этом он стал бы еще менее понятным и более "хрупким". (Как любой запрос, использующий подсказки: подсказка — это всего лишь предложение, и оптимизатор может его проигнорировать). Запрос с аналитическими функциями, определенно, выигрывает как по производительности, так и по простоте.

Теперь переходим ко второму вопросу: выдать до трех человек с максимальными зарплатами.

```
scott@TKYTE816> select *
  2  from (select deptno, ename, sal,
  3        count(*) over (partition by deptno
  4                       order by sal desc
  5                       range unbounded preceding)
  6         cnt from t)
  7  where cnt <= 3
  8  order by deptno, sal desc
  9  /
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.01	0	0	0	0
Execute	2	0.00	0.00	0	0	0	0
Fetch	11	0.02	0.12	15	10	17	150
total	**14**	**0.03**	**0.13**	**15**	**10**	**17**	**150**

```
Misses in library cache during parse: 0
Optimizer goal: CHOOSE
Parsing user id: 54
```

Rows	Row Source Operation
150	VIEW
1000	WINDOW SORT
1000	TABLE ACCESS FULL T

```
**********************************

scott@TKYTE816> select deptno, ename, sal
  2  from t e1
  3  where (select count(*)
  4         from t e2
  5        where e2.deptno = e1.deptno
  6         and e2.sal >= e1.sal) <= 3
```

```
  7  order by deptno, sal desc
  8  /
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.01	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	11	0.60	0.66	0	4010	4012	150
total	**13**	**0.61**	**0.67**	**0**	**4010**	**4012**	**150**

```
Misses In library cache during parse: 0
Optimizer goal: CHOOSE
Parsing user id: 54
```

```
Rows     Row Source Operation

    150  SORT ORDER BY
    150   FILTER
   1001    TABLE ACCESS FULL T
   2000    SORT AGGREGATE
  10827     INDEX FAST FULL SCAN (object id 27867)
```

И на этот раз результаты говорят сами за себя: 0,03 секунды процессорного времени и 27 логических операций ввода-вывода по сравнению с 0,61 секунды процессорного времени и более чем 8000 логических операций ввода-вывода. Снова явное преимущество имеет запрос, использующий аналитические функции. Причина и на этот раз в том, что при отсутствии аналитических функций для каждой строки базовой таблицы приходится выполнять коррелированный подзапрос. Этот подзапрос подсчитывает количество записей в том же отделе для сотрудников с такой же или более высокой зарплатой. Выбираются только записи, для которых это количество меньше или равно 3. Результаты запросов одинаковы, но используемые во время выполнения ресурсы принципиально различны. Составить оба запроса одинаково несложно, но производительность при использовании аналитических функций несравнимо выше.

Наконец, необходимо получить для каждого отдела первых три сотрудника с наибольшими зарплатами. Результаты следующие:

```
scott@TKYTE816> select deptno, ename, sal
  2       from t el
  3       where (select count(*)
  4              from t e2
  5             where e2.deptno = el.deptno
  6             and e2.sal >= el.sal ) <=3
  7       order by deptno, sal desc
  8  /
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	2	0.00	0.00	0	0	0	0
Fetch	11	0.00	0.12	14	10	17	150
total	**14**	**0.00**	**0.12**	**14**	**10**	**17**	**150**

```
Misses in library cache during parse: 0
Optimizer goal: CHOOSE
Parsing user id: 54

Rows     Row Source Operation

    150  VIEW
   1000   WINDOW SORT
   1000    TABLE ACCESS FULL T
************************************

scott@TKYTE816> select deptno, ename, sal
  2  from t e1
  3  where (select count(*)
  4         from t e2
  5         where e2.deptno = e1.deptno
  6         and e2.sal >= e1.sal
  7         and (e2.sal > e1.sal OR e2.rowid > e1.rowid)) < 3
  8  order by deptno, sal desc
  9  /
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	11	0.88	0.88	0	4010	4012	150
total	**13**	**0.88**	**0.88**	**0**	**4010**	**4012**	**150**

```
Misses in library cache during parse: 0
Optimizer goal: CHOOSE
Parsing user id: 54

Rows     Row Source Operation

    150  SORT ORDER BY
    150   FILTER
   1001    TABLE ACCESS FULL T
   2000    SORT AGGREGATE
   9827     INDEX FAST FULL SCAN (object id 27867)
```

И на этот раз производительность запросов несравнима. Версия с аналитической функцией во много раз превосходит по производительности реляционный запрос. Кроме того, версию с использованием аналитических функций в данном случае написать намного проще. Созданный коррелированный подзапрос весьма сложен. Необходимо подсчитать количество сотрудников в отделе, у которых зарплата больше или равна зарплате в текущей записи. Кроме того, если зарплата не больше зарплаты в текущей записи (совпадает с ней), такую запись можно учитывать, только если значение **ROWID** (или любого столбца с уникальными значениями) больше, чем в текущей записи. Это гарантирует, что строки не будут считаться дважды, а каждый раз будет выбираться другой набор строк.

Во всех случаях оказалось, что аналитические функции не только упрощают написание сложных запросов, но и могут существенно повысить производительность. Они позволяют делать то, что не стоит запрашивать с помощью "чистого" языка SQL из-за неэффективности выполнения.

Запрос с транспонированием

При *запросе с транспонированием* (опорный запрос — pivot query) берутся данные вида:

```
C1      C2      C3

a1      b1      x1
a1      b1      x2
a1      b1      x3
```

и выдаются в следующем виде:

```
Cl      C2      C3(l)   C3(2)   C3(3)

a1      b1      x1      x2      x3
```

Этот запрос преобразует строки в столбцы. Например, можно выдать должности сотрудников отдела в виде столбцов:

```
DEPTNO      JOB_1      JOB_2      JOB_3

        10 CLERK      MANAGER    PRESIDENT
        20 ANALYST    ANALYST    CLERK
        30 CLERK      MANAGER    SALESMAN
```

а не в виде строк:

```
DEPTNO      JOB

        10 CLERK
        10 MANAGER
        10 PRESIDENT
        20 ANALYST
        20 CLERK
        20 MANAGER
        30 CLERK
        30 MANAGER
        30 SALESMAN
```

Я представлю два примера запросов с транспонированием. Первый — разновидность описанного выше запроса трех сотрудников с максимальными зарплатами. Второй пример показывает, как транспонировать любое результирующее множество, и дает шаблон необходимых для этого действий.

Предположим, необходимо выдать фамилии сотрудников отдела с тремя наибольшими зарплатами в виде **столбцов.** Запрос должен возвращать ровно одну строку для каждого отдела, причем в строке должно быть 4 столбца: номер отдела **(DEPTNO),** фамилия сотрудника с наибольшей зарплатой в отделе, фамилия сотрудника со следующей

по величине зарплатой и т.д. С помощью новых аналитических функций это сделать просто (а до их появления — практически невозможно):

```
ops$tkyte@DEV816> select deptno,
  2            max(decode(seq,1,ename,null)) highest_paid,
  3            max(decode(seq,2,ename,null)) second_highest,
  4            max(decode(seq,3,ename,null)) third_highest
  5    from (SELECT deptno, ename,
  6                 row_number() OVER
  7                      (PARTITION BY deptno
  8                      ORDER BY sal desc NULLS LAST) seq
  9           FROM emp)
 10    where seq <= 3
 11    group by deptno
 12  /

    DEPTNO HIGHEST_PA SECOND_HIG THIRD_HIGH

        10 KING       CLARK      MILLER
        20 SCOTT      FORD       JONES
        30 BLAKE      ALLEN      TURNER
```

Мы создали внутреннее результирующее множество, где сотрудники отделов пронумерованы по убыванию зарплат. Функция decode во внешнем запросе оставляет только строки со значениями номеров 1, 2 или 3 и присваивает взятые из них фамилии соответствующему столбцу. Конструкция GROUP BY позволяет избавиться от лишних строк и получить сжатый результат. Возможно, понять, что я имею в виду, проще, если сначала посмотреть на результирующее множество запроса без конструкций GROUP BY и **MAX**:

```
scott@TKYTE816> select deptno,
  2    decode(seq,1,ename,null) highest_paid,
  3    decode(seq,2,ename,null) second_highest,
  4    decode(seq, 3,ename,null) third_highest
  5  from (select deptno, ename,
  6        row_number() over
  7           (partition by deptno
  8           order by sal desc nulls last)
  9        seq from emp)
 10  where seq <= 3
 11  /

    DEPTNO HIGHEST_PA SECOND_HIG THIRD_HIGH

        10 KING
        10            CLARK
        10                       MILLER

        20 SCOTT
        20            FORD
        20                       JONES

        30 ALLEN
        30            BLAKE
```

```
30                    MARTIN

9 rows selected.
```

Функция агрегирования **MAX** будет применяться конструкцией группировки **GROUP BY** по столбцу **DEPTNO**. Значение в столбце **HIGHEST_PAID** для отдела только в одной строке будет непустым — в остальных строках этот столбец всегда будет иметь значение **NULL**. Функция **MAX** будет выбирать только строку с непустым значением. Поэтому сочетание группирования и функции **MAX** позволит, убрав значения **NULL**, "свернуть" результирующее множество и получить желаемый результат.

Если есть таблица T со столбцами C1 и C2 и необходимо получить результат вида:

C1 C2(1) C2(2) .... C2(N),

где столбец C1 должен присутствовать **во всех строках** (значения выдаются по направлению к концу страницы), а столбец C2 должен быть транспонирован так, чтобы он представлялся в виде **строк** (значения C2 выдаются по направлению к концу строки, они становятся столбцами, а не строками), надо создать такой запрос:

```
select c1,
  max(decode(rn,1,c2,null)) c2_1,
  max(decode(rn,2,c2,null)) c2_2,

  max(decode(rn,N,c2,null)) c2_N
from (select c1, c2,
      row_number() over (partition by C1
                         order by <столбцы>)
      rn from T
      <условие>)
group by C1
```

В представленном выше примере в качестве C1 использовался столбец **DEPTNO,** а в качестве **C2 — ENAME.** Поскольку упорядочение выполнялось по критерию **SAL DESC**, первые три полученные строки соответствовали трем наиболее высокооплачиваемым сотрудникам соответствующего отдела (напоминаю: если максимальные зарплаты получало четыре человека, одного из них мы теряем).

Второй пример: транспонировать результирующее множество. Рассмотрим более общий случай, когда *опорный* (отсюда и второе название запроса — *опорный)* столбец, **C1**, и транспонируемый столбец, **C2**, представляют собой наборы столбцов. Решение очень похоже на то, что представлено выше. Предположим, необходимо для каждого отдела и должности выдать фамилии и зарплаты сотрудников. При этом в отчете фамилии и соответствующие зарплаты должны выдаваться **в строке,** как столбцы. Кроме того, в строке сотрудников надо упорядочивать слева направо по возрастанию зарплат. Для решения этой проблемы необходимо выполнить следующее:

```
scott@TKYTE816> select max(count(*)) from emp group by deptno, job;

MAX(COUNT(*))

            4
```

В результате мы получаем количество столбцов. Теперь можно создавать запрос:

```
scott@TKYTE816> select deptno, job,
  2   max(decode(rn, 1, ename, null)) ename_1,
  3   max(decode(rn, 1, sal, null)) sal_1,
  4   max(decode(rn, 2, ename, null)) ename_2,
  5   max(decode(rn, 2, sal, null)) sal_2,
  6   max(decode(rn, 3, ename, null)) ename_3,
  7   max(decode(rn, 3, sal, null)) sal_3,
  8   max(decode(rn, 4, ename, null)) ename_4,
  9   max(decode(rn, 4, sal, null)) sal_4
 10  from (select deptno, job, ename, sal,
 11        row_number() over (partition by deptno, job
 12                           order by sal, ename)
 13         rn from emp)
 14  group by deptno, job
 15  /
```

DEPTNO	JOB	ENAME_1	SAL_1	ENAME_2	SAL_2	ENAME_3	SAL_3	ENAME_	SAL_4
10	CLERK	MILLER	1300						
10	MANAGER	CLARK	2450						
10	PRESIDENT	KING	5000						
20	ANALYST	FORD	3000	SCOTT	3000				
20	CLERK	SMITH	800	ADAMS	1100				
20	MANAGER	JONES	2975						
30	CLERK	JAMES	99						
30	MANAGER	BLAKE	99						
30	SALESMAN	ALLEN	99	MARTIN	99	TURNER	99	WARD	99

```
9 rows selected.
```

Ранее в этой главе мы установили значение зарплаты 99 для сотрудников отдела 30. Для транспонирования произвольного результирующего множества можно пойти еще дальше. Если имеется набор столбцов **C1, C2, C3, ... CN** и значения столбцов **C1 ... Cx** должны выдаваться **во всех строках,** а значения столбцов **Cx+1** ... **CN — в виде столбцов каждой строки,** запрос будет иметь такой синтаксис:

```
Select C1, C2, ... CX,
  max(decode(rn,1,C{X+1},null)) cx+1_1,.. .rnax(decode(rn,1,CN,null)) CN_1
  max(decode(rn,2,C{X+1},null)) cx+1_2,...max(decode(rn,1,CN,null)) CN_2

  max(decode(rn,N,c{X+1},null)) cx+1_N,...max(decode(rn,1,CN,null)) CN_N
from (select C1, C2, ... CN
      row_number() over (partition by C1, C2, ... CX
                         order by <столбцы>)
      rn from T
      <условие>)
group by C1, C2, ... CX
```

В предыдущем примере в качестве C1 использовался столбец **DEPTNO,** в качестве **C2 — JOB, C3** представлял столбец **ENAME,** а **C4 — SAL**.

Для создания подобного запроса надо знать **максимальное** количество строк, которое может быть в фрагменте. Оно определяет количество генерируемых столбцов. В SQL

необходимо знать количество выбираемых столбцов, т.к. в противном случае мы не сможем транспонировать результирующее множество. Таким образом, можно привести еще более общий пример создания запроса с транспонированием. Если заранее, до выполнения, общее количество столбцов не известно, придется использовать динамический SQL, чтобы справиться с переменным списком выбора в операторе **SELECT**. Для демонстрации этого можно написать PL/SQL-процедуру; в результате мы получим универсальную процедуру для транспонирования любого результирующего множества. Эта процедура (я поместил ее в пакет) будет иметь следующую спецификацию:

```
scott@TKYTE816> create or replace package my_pkg
  2  as
  3    type refcursor is ref cursor;
  4    type array is table of varchar2(30);
  5    procedure   pivot(p_max_cols      in number   default NULL,
  6                      p_max_cols_query in varchar2 default NULL,
  7                        p_query         in varchar2,
  8                        p_anchor        in array,
  9                        p_pivot         in array,
 10                        p_cursor in out refcursor);
 12  end;

Package created.
```

Необходимо задать значения для параметра **P_MAX_COLS** или для параметра **P_MAX_COLS_QUERY**. Для создания SQL-оператора необходимо знать количество столбцов в запросе, и эти параметры позволят создать запрос с соответствующим количеством столбцов. Задавать этим параметрам надо значение, полученное в результате выполнения такого запроса:

```
scott@TKYTE816> select max(count(*)) from emp group by deptno, job;
```

Он возвращает количество различных значений в **строках,** которые мы хотим транспонировать. Можно либо получить это количество с помощью подобного запроса, либо просто ввести его, если оно заранее известно.

Параметр **P_QUERY** — это запрос, собирающий данные. Для представленного выше примера можно передать следующий запрос:

```
 10  from (select deptno, job, ename, sal,
 11        row_number() over (partition by deptno, job
 12                           order by sal, ename)
 13         rn from emp)
```

Следующие два параметра — массивы имен столбцов. Параметр **P_ANCHOR** указывает, значения каких столбцов остаются в строках, а параметр **P_PIVOT** перечисляет столбцы, значения которых выносятся в строки. В рассмотренном ранее примере **P_ANCHOR = ('DEPTNO', 'JOB'),** а **P_PIVOT = ('ENAME','SAL')**. Отвлечемся ненадолго от нашей темы и рассмотрим, как может выглядеть вызов процедуры транспонирования результирующего множества:

```
scott@TKYTE816> variable x refcursor
scott@TKYTE816> set autoprint on
```

```
scott@TKYTE816> begin
  2  my_pkg.pivot
  3  (p_max_cols_query => 'select max(count(*)) from emp
  4                        group by deptno,job',
  5  p_query => 'select deptno, job, ename, sal,
  6   row_number() over (partition by deptno, job
  7                      order by sal, ename)
  8   rn from emp a',
  9
 10     p_anchor => my_pkg.array('DEPTNO','JOB'),
 11    p_pivot    => my_pkg.array('ENAME', 'SAL'),
 12     p_cursor => :x);
 13  end;

PL/SQL procedure successfully completed.

DEPTNO JOB        ENAME_ SAL_1 ENAME_2 SAL_2 ENAME_3  SAL_3 ENAME_ SAL_4

    10 CLERK      MILLER  1300
    10 MANAGER    CLARK   2450
    10 PRESIDENT  KING    5000
    20 ANALYST    FORD    3000 SCOTT    3000
    20 CLERK      SMITH    800 ADAMS    1100
    20 MANAGER    JONES   2975
    30 CLERK      JAMES     99
    30 MANAGER    BLAKE     99
    30 SALESMAN   ALLEN     99 MARTIN     99 TURNER      99 WARD      99

9 rows selected.
```

Как видите, запрос динамически переписан на базе разработанного универсального шаблона. Реализация тела пакета достаточно проста:

```
scott@TKYTE816> create or replace package body my_pkg
  2  as
  3
  4  procedure pivot(p_max_cols       in number   default null,
  5                  p_max_cols_query in varchar2 default null,
  6                  p_query          in varchar2,
  7                  p_anchor         in array,
  8                  p_pivot          in array,
  9                  p_cursor in out refcursor)
 10  as
 11      l_max_cols number;
 12      l_query    long;
 13      l_cnames   array;
 14  begin
 15    -- определяем количество столбцов, которые надо возвращать
 16    -- мы либо ЗНАЕМ его, либо получаем запрос, с помощью которого его
       -- можно узнать
 17    if (p_max_cols is not null)
 18    then
 19          l_max_cols := p_max_cols;
 20    elsif (p_max_cols_query is not null)
```

```
21    then
22          execute Immediate p_max_cols_query into l_max_cols;
23    else
24          raise_application_error(-20001, 'Не могу определить
              максимальное количество столбцов');
25    end if;
26
27
28    -- Теперь создаем запрос, который позволяет ответит на
      -- поставленный вопрос...
29    -- Начинаем со столбцов C1, C2, ... CX:
30
31    l_query := 'select ';
32    for i in 1 .. p_anchor.count
33
34    loop
35      l_query := l_query || p_anchor(i) || ',';
36    end loop;
37
38    -- Теперь добавляем транспонируемые столбцы C{x+1}... CN:
39    -- Формат: "max(decode(rn,1,C{X+1},null)) cx+1_1"
40
41    for i in 1 .. l_max_cols
42    loop
43      for j in 1 .. p_pivot.count
44         loop
45           l_query := l_query ||
46                      'max(decode(rn,'Hill','||
47                    p_pivot(j)||',null)) ' ||
48                   p_pivot(j) || '_' || i || ',';
49         end loop;
50    end loop;
51
52     -- Теперь просто добавляем исходный запрос
53
54   l_query := rtrim(l_query,',') || ' from (' || p_query || ') group by ';
55
56     -- а затем — столбцы, по которым надо группировать...
57
58    for i in 1 .. p_anchor.count
59    loop
60        l_query := l_query || p_anchor(i) || ',';
61    end loop;
62    l_query := rtrim(l_query,',');
63
64     -- и возвращаем курсор для результирующего множества
65
66    execute immediate 'alter session set cursor_sharing=force';
67    open p_cursor for l_query;
68    execute immediate 'alter session set cursor_sharing=exact';
69    end;
```

```
70
71  end;
72  /

Package body created.
```

Понадобилось несколько строковых функций для перезаписи запроса и динамическое открытие курсорной переменной (REF CURSOR). Поскольку вполне вероятно, что в условии запроса есть константы, мы включаем опцию cursor_sharing перед анализом запроса, чтобы принудительно использовались связываемые переменные, а затем отключаем ее. (Подробнее об этом см. в главе 10, посвященной настройке производительности). В результате получаем полностью проанализированный запрос, готовый для извлечения данных через курсорную переменную.

Доступ к строкам вокруг текущей строки

Часто необходимо обращаться к данным не только в текущей строке, но и в ближайших предыдущих или последующих. Предположим, необходимо создать отчет, в котором по отделам были бы представлены все сотрудники, причем, для каждого сотрудника выдана дата его приема на работу, за сколько дней до этой даты последний раз принимали сотрудника на работу, и через сколько дней после этого приняли на работу следующего сотрудника. Написание подобного запроса с помощью "чистого" языка SQL — чрезвычайно сложная задача. Более того, производительность полученного запроса вызывает сомнения. В прошлом я либо пытался применять прием "select из select", либо писал PL/SQL-функцию, которая по данным из текущей строки находила предыдущую и следующую строки данных. Это работало, но очень много времени уходило на разработку запроса (приходилось писать больше кода); кроме того, расходовалось большое количество ресурсов при его выполнении.

С помощью аналитических функций это делается быстро и эффективно. Соответствующий запрос будет выглядеть так:

```
scott@TKYTE816> select deptno, ename, hiredate,
  2       lag(hiredate, 1, null) over (partition by deptno
  3                                    order by hiredate, ename) last_hire,
  4       hiredate - lag(hiredate, 1, null)
  5                    over (partition by deptno
  6                          order by hiredate, ename) days_last,
  7       lead(hiredate, 1, null)
  8         over (partition by deptno
  9               order by hiredate, ename) next_hire,
 10       lead(hiredate, 1, null)
 11         over (partition by deptno
 12               order by hiredate, ename) - hiredate days_next
 13  from emp
 14  order by deptno, hiredate
 15  /

    DEPTNO ENAME    HIREDATE  LAST_HIRE  DAYS_LAST NEXT_HIRE  DAYS_NEXT

        10 CLARK    09-JUN-81                      17-NOV-81        161
           KING     17-NOV-81 09-JUN-81       161  23-JAN-82         67
```

```
         MILLER    23-JAN-82 17-NOV-81          67

      20 SMITH     17-DEC-80                       02-APR-81         106
         JONES     02-APR-81      17-DEC-80       106 03-DEC-81         245
         FORD      03-DEC-81 02-APR-81            245 09-DEC-82         371
         SCOTT     09-DEC-82 03-DEC-81            371 12-JAN-83          34
         ADAMS     12-JAN-83 09-DEC-82             34

      30 ALLEN     20-FEB-81                       22-FEB-81           2
         WARD      22-FEB-81 20-FEB-81             2 01-MAY-81          68
         BLAKE     01-MAY-81 22-FEB-81            68 08-SEP-81         130
         TURNER    08-SEP-81 01-MAY-81           130 28-SEP-81          20
         MARTIN    28-SEP-81 08-SEP-81            20 03-DEC-81          66
         JAMES     03-DEC-81 28-SEP-81            66

14 rows selected.
```

Функции LEAD и LAG можно рассматривать как способы индексации в пределах группы. С помощью этих функций можно обратиться к любой отдельной строке. Обратите внимание: в представленных выше результатах запись для сотрудника KING включает данные (выделены полужирным) из предыдущей строки (LAST_HIRE) и последующей (NEXT_HIRE). Можно получить поля предыдущих или последующих записей в упорядоченном фрагменте.

Прежде чем подробно рассматривать функции LAG и LEAD, я хотел бы сравнить этот запрос с аналогичным по результатам запросом, в котором не используются аналитические функции. Для этого я создам необходимые индексы по таблице, чтобы максимально быстро получать ответ:

```
scott@TKYTE816> create table t
  2  as
  3  select object_name ename,
  4         created hiredate,
  5             mod(object_id,50)  deptno
  6  from all_objects
  7  /

Table created.

scott@TKYTE816> alter table t modify deptno not null;

Table altered.

scott@TKYTE816> create index t_idx on t(deptno,hiredate,ename)
  2  /

Index created.

scott@TKYTE816> analyze table t
  2  compute statistics
  3  for table
  4  for all indexes
  5  for all indexed columns
  6  /

Table analyzed.
```

Я даже добавил в индекс столбец ENAME, чтобы при выполнении запроса достаточно было обращения **только** к индексу, без обращения к таблице по значению **ROWID.** Запрос с аналитической функцией демонстрирует следующую производительность:

```
scott@TKYTE816> select deptno, ename, hiredate,
  2  lag(hiredate, 1, null) over (partition by deptno
  3                               order by hiredate, ename) last_hire,
  4     hiredate - lag(hiredate, 1, null)
  5      over (partition by deptno
  6           order by hiredate, ename) days_last,
  7     lead(hiredate, 1, null)
  8      over (partition by deptno
  9           order by hiredate, ename) next_hire,
 10     lead(hiredate, 1, null)
 11      over (partition by deptno
 12           order by hiredate, ename) - hiredate days_next
 13  from emp
 14  order by deptno, hiredate
 15  /
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.01	0	0	0	0
Execute2	0.00	0.00	0	0	0	0	
Fetch	1313	0.72	1.57	142	133	2	19675
total	**1316**	**0.73**	**1.58**	**142**	**133**	**2**	**19675**

```
Misses in library cache during parse: 0
Optimizer goal: FIRST_ROWS
Parsing user id: 54
```

Rows	Row Source Operation
19675	WINDOW BUFFER
19675	INDEX FULL SCAN (object id 27899)

Сравним с эквивалентным запросом, где аналитические функции не используются:

```
scott@TKYTE816> select deptno, ename, hiredate,
  2  hiredate-(select max(hiredate)
  3            from t e2
  4            where e2.deptno = e1.deptno
  5            and e2.hiredate < e1.hiredate) last_hire,
  6  hiredate-(select max(hiredate)
  7            from t e2
  8            where e2.deptno = e1.deptno
  9            and e2.hiredate < e1.hiredate) days_last,
 10           (select man(hiredate)
 11            from t e3
 12            where e3.deptno = e1.deptno
 13            and e3.hiredate > e1.hiredate) next_hire,
 14           (select min(hiredate)
 15            from t e3
 16            where e3.deptno = e1.deptno
```

```
 17           and e3.hiredate > e1.hiredate) - hiredate days_next
 18    from t e1
 19  order by deptno, hiredate
 20  /
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.01	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	1313	2.48	2.69	0	141851	0	19675
total	**1315**	**2.49**	**2.70**	**0**	**141851**	**0**	**19675**

```
Misses in library cache during parse: 0
Optimizer goal: FIRST_ROWS
Parsing user id: 54
```

Rows Row Source Operation

```
  19675  INDEX FULL SCAN (object id 27899)
```

Производительность этих двух запросов существенно отличается. Сравните: 135 логических операций ввода-вывода и 141000; 0,73 секунды процессорного времени и 2,49. Запрос с аналитической функцией и в этом случае оказался намного эффективнее. Учтите также сложность текста запросов. Мне кажется, запрос с помощью функций LAG и LEAD не только проще написать, но и понять впоследствии, что выбирается. Прием "select из select" — хороший трюк, но такой код сложнее придумать, а при чтении полученного запроса часто очень трудно понять, что он выбирает. Чтобы восстановить логику второго запроса, придется намного больше думать.

Теперь давайте более детально рассмотрим функции **LAG** и **LEAD**. Эти функции принимают три аргумента:

```
lag(Arg1, Arg2, Arg3)
```

- **Arg1** — выражение, которое надо вернуть на основе другой строки.
- **Arg2** — смещение требуемой строки в группе относительно текущей. Смещение задается как положительное целое число. В случае функции **LAG** берется соответствующая смещению предыдущая строка, а в случае функции **LEAD** — следующая. Этот аргумент имеет стандартное значение 1.
- **Arg3** — возвращаемое значение в том случае, если смещение, заданное аргументом Arg2, выводит за границу группы. Например, первая строка в каждой группе не имеет предыдущей, так что значение функции **LAG(..., 1)** для этой строки определить нельзя. Можно возвращать стандартное значение **NULL** или указать значение явно. Следует учитывать, что окна для функций **LAG** и **LEAD** не используются — можно задавать конструкции **PARTITION BY** и **ORDER BY**, но не **ROWS** или **RANGE**.

Итак, в нашем примере:

```
  4       hiredate - lag(hiredate, 1, null)
  5                    over (partition by deptno
  6                      order by hiredate, ename) days_last,
```

функция LAG использовалась для поиска предыдущей строки, поскольку в качестве второго параметра передавалось значение 1 (если предыдущей записи нет, возвращается значение NULL). Мы фрагментировали данные по столбцу DEPTNO, так что каждый отдел просматривается независимо от остальных. Полученный фрагмент мы упорядочили по значению столбца HIREDATE, так что вызов LAG(HIREDATE, 1, NULL) возвращает максимальное значение HIREDATE, меньшее соответствующего значения в текущей строке.

Проблемы

С аналитическими функциями у меня почти не было проблем. Они позволяют получать ответ на абсолютно новые запросы намного эффективнее, чем до их появления. Если разобраться в синтаксисе, открываются безграничные возможности. Не часто бывает, когда результат можно получить практически даром, но с аналитическими функциями все обстоит, похоже, именно так. Следует, однако, помнить о четырех потенциальных проблемах.

Аналитические функции в PL/SQL

При попытке использования аналитических функций в коде на языке PL/SQL могут возникать ошибки. Если взять простой запрос и поместить его в PL/SQL-блок:

```
scott@TKYTE816> variable x refcursor
scott@TKYTE816> set autoprint on
scott@TKYTE816>begin
  2  open :x for
  3  select mgr, ename,
  4         row_number() over (partition by mgr
  5                            order by ename)
  6          rn from emp;
  7  end;
  8  /
           row_number() over (partition by mgr
                              *
ERROR at line 5:
ORA-06550: line 5, column 31:
PLS-00103: Encountered the symbol "(" when expecting one of the
following:
, from into bulk
```

синтаксический анализатор PL/SQL его не воспримет. Анализатор SQL-операторов, используемый в PL/SQL, еще не понимает (в версиях Oracle 8i — *прим. научн. ред.)* синтаксис вызова аналитических функций. Сталкиваясь с подобными проблемами (есть и другие конструкции, не воспринимаемые синтаксическим анализатором PL/SQL), я использую динамически открываемую курсорную переменную. Реализация показанного выше запроса в этом случае может выглядеть так:

```
scott@TKYTE816> variable x refcursor
scott@TKYTE816> set autoprint on
```

```
scott@TKYTE816> begin
  2  open :x for
  3  'select mgr, ename,
  4         row_number() over (partition by mgr
  5                            order by ename)
  6   rn from emp';
  7  end;
  8  /

PL/SQL procedure successfully completed.

       MGR ENAME                RN

      7566 FORD                  1
      7566 SCOTT                 2
      7698 ALLEN                 1
      7698 JAMES                 2
      7698 MARTIN                3
      7698 TURNER                4
      7698 WARD                  5
      7782 MILLER                1
      7788 ADAMS                 1
      7839 BLAKE                 1
      7839 CLARK                 2
      7839 JONES                 3
      7902 SMITH                 1
           KING                  1

14 rows selected.
```

Мы "обманули" синтаксический анализатор PL/SQL, не разрешая ему анализировать конструкции, которые он не понимает, в данном случае — вызов функции **ROW_NUMBER().** Для этого достаточно использовать динамически открываемые курсорные переменные. После открытия они работают аналогично **обычным** курсорам: из них извлекаются данные, потом курсорные переменные закрываются и т.д., но PL/SQL-машина не пытается анализировать операторы ни во время компиляции, ни при выполнении, поэтому можно использовать новые синтаксические конструкции языка SQL.

Можно также создать представление на базе запроса с аналитическими функциями, а затем обращаться в PL/SQL-блоке к этому представлению. Например:

```
scott@TKYTE816> create or replace view
  2  emp_view
  3  as
  4  select mgr, ename,
  5         row_number() over (partition by mgr
  6                            order by ename) rn
  7    from emp
  8  /

View created.

scott@TKYTE816> begin
  2  open :x for
  3  select mgr, ename, rn
```

```
  4    from emp_view;
  5  end;
  6  /

PL/SQL procedure successfully completed.

      MGR ENAME                RN

     7566 FORD                  1
     7566 SCOTT                 2
```

Аналитические функции в конструкции WHERE

Следует учитывать, что аналитические функции применяются по ходу выполнения запроса почти в самом конце (после них обрабатывается только окончательная конструкция ORDER BY). Это означает, что аналитические функции нельзя непосредственно использовать в условиях (т.е. применять в конструкциях WHERE и HAVING). Если необходимо включать данные в результирующее множество на основе результатов аналитической функции, придется использовать подставляемое представление. Аналитические функции могут использоваться только в списке выбора или в конструкции ORDER BY запроса.

В этой главе приводилось много примеров использования подставляемых представлений, в частности в разделе, посвященном выбору первых N строк. Например, чтобы найти группу сотрудников каждого отдела с тремя наибольшими зарплатами, мы выполняли следующий запрос:

```
scott@TKYTE816> select *
  2  from (select deptno, ename, sal,
  3        dense_rank() over (partition by deptno
  4                           order by sal desc) dr
  5           from emp)
  6  where dr <= 3
  7  order by deptno, sal desc
  8  /
```

Поскольку функцию **DENSE_RANK** нельзя использовать в конструкции **where** непосредственно, приходится скрывать ее в подставляемом представлении под псевдонимом **DR**, чтобы в дальнейшем можно было использовать столбец **DR** в условии для получения необходимых строк. Такой прием часто используется при работе с аналитическими функциями.

Значения NULL и сортировка

Значения NULL могут влиять на результат работы аналитических функций, особенно при использовании сортировки по убыванию. По умолчанию значения NULL считаются больше любых других значений. Рассмотрим следующий пример:

```
scott@TKYTE816> select ename, comm from emp order by comm desc;

ENAME                COMM

SMITH
JONES
```

```
CLARK
BLAKE
SCOTT
KING
JAMES
MILLER
FORD
ADAMS
MARTIN          1400
WARD             500
ALLEN            300
TURNER             0

14 rows selected.
```

Выбрав первые N строк, получим:

```
scott@TKYTE816> select ename, comn, dr
  2  from (select ename/ comm,
  3        denserankO over (order by comm desc)
  4        dr from emp)
  5  where dr <= 3
  6  order by comm
  8  /

ENAME             COM4          DR

SMITH                            1
JONES                            1
CLARK                            1
BLAKE                            1
SCOTT                            1
KING                             1
JAMES                            1
MILLER                           1
FORD                             1
ADAMS                            1
MARTIN          1400             2
HARD             500             3

12 rows selected.
```

Хотя формально это верно, но вряд ли соответствует желаемому результату. Значения **NULL** либо вообще не должны учитываться, либо интерпретироваться как "наименьшие" в данном случае. Поэтому надо либо исключить значения **NULL** из рассмотрения, добавив условие **where comm is not null**:

```
scott@TKYTE816> select ename, comm, dr
  2  from (select ename, comm,
  3        dense_rank() over (order by comm desc)
  4        dr from emp
  5        where comm is not null)
  6  where dr <= 3
  7  order by comm desc
  8  /
```

```
ENAME          COMM          OR

MARTIN         1400           1
HARD            500           2
ALLEN           300           3
```

либо использовать NULLS LAST в конструкции ORDER BY:

```
scott@TKYTE816> select ename, comm, dr
  2  from (select ename, comm,
  3        dense_rank() over (order by comm desc nulls last)
  4         dr from emp
  5        where comm is not null)
  6  where dr <= 3
  7  order by comm desc
  8  /

ENAME          COM4          DR

MARTIN         1400           1
WARD            500           2
ALLEN           300           3
```

Следует помнить, что NULLS LAST можно указывать и в обычных конструкциях ORDER BY, а не только при вызове аналитических функций.

Производительность

До сих пор все, что удалось узнать об аналитических функциях, свидетельствует о них как об универсальном средстве повышения производительности. Однако при неправильном использовании они могут отрицательно повлиять на производительность.

При использовании этих функций надо опасаться видимой легкости, с которой они позволяют сортировать и фильтровать множества немыслимыми в стандартном языке SQL способами. Каждый вызов аналитической функции в списке выбора оператора SELECT может использовать свои фрагменты, окна и порядок сортировки. Если они несовместимы (не являются подмножествами друг друга), может выполняться огромный объем сортировки и фильтрования. Например, ранее мы выполняли следующий запрос:

```
ops$tkyte@DEV816> select ename, deptno,
  2       sum(sal) over (order by ename, deptno) sum_ename_deptno,
  3       sum(sal) over (order by deptno, ename) sum_deptno_ename
  4    from emp
  5   order by ename, deptno
  6  /
```

В этом запросе имеются три конструкции ORDER BY, то есть может потребоваться три сортировки. Две сортировки можно объединить, поскольку они выполняются по одним и тем же столбцам, но третью — придется выполнять отдельно. Это — не повод для беспокойства или отказа от использования аналитических функций. Просто надо это учитывать. С помощью аналитических функций можно так же легко написать запрос, использующий все ресурсы компьютера, как и запросы, элегантно и эффективно решающие сложные задачи.

Резюме

В этой главе мы подробно рассмотрели синтаксис и возможности аналитических функций. Было показано, с какой легкостью они позволяют решать типичные задачи вроде вычисления частичных сумм, транспонирования результирующих множеств, доступа из текущей строки к "соседним" строкам и т.д. Аналитические функции дают широкий спектр потенциальных возможностей для запросов.

13

Материализованные представления

Материализованные представления — средство повышения производительности для хранилищ данных и систем поддержки принятия решений, которое многократно ускоряет выполнение запросов, обращающихся к большому количеству (сотням тысяч или миллионам) записей. Говоря упрощенно, они позволяют за секунды (и даже доли секунд) выполнять запросы к терабайтам данных. Это достигается за счет прозрачного использования заранее вычисленных итоговых данных и результатов соединений таблиц. Предварительно вычисленные итоговые данные обычно имеют очень небольшой объем по сравнению с исходными данными.

Предположим, в компании имеется база данных продаж, в которую загружены сведения о миллионах заказов, и необходимо проанализировать продажи по регионам (весьма типичный запрос). Будут просмотрены все записи, данные — агрегированы по регионам с выполнением необходимых вычислений. С помощью материализованного представления можно сохранить итоговые данные продаж по регионам и обеспечить автоматическую поддержку этих данных системой. При наличии десяти регионов продаж итоговые данные будут состоять из десяти записей, так что мы будем обращаться не к миллиону фактических записей, а только к десяти. Более того, при выполнении несколько измененного запроса, например об объеме продаж по определенному региону, ответ на него тоже можно получить по этому материализованному представлению.

В этой главе мы разберемся, что такое материализованные представления, их возможности и, самое главное, как они устроены. Я покажу, как обеспечить использование созданного материализованного представления **всеми** запросами, для которых оно позволяет получить ответ (иногда **очевидно,** что сервер Oracle мог бы использовать ма-

териализованное представление, но не делает этого из-за отсутствия важной информации). В частности, будет:

- рассмотрен пример, демонстрирующий возможности материализованных представлений и позволяющий решить, пригодятся ли они вам;
- описаны параметры и привилегии, которые необходимо установить для использования материализованных представлений;
- показано на примерах использование *измерений* (dimensions) и требований целостности, позволяющих серверу определить, когда для ответа на запрос оправданно применение материализованного представления;
- описано использование пакета DBMS_OLAP для анализа представлений;
- и в завершение описаны две проблемы, которые надо учитывать при использовании материализованных представлений.

Предыстория

Управление итоговыми таблицами — еще одна возможность материализованного представления — уже некоторое время использовалось в инструментальных средствах типа Oracle Discoverer (средство построения произвольных запросов и отчетов). С помощью Discoverer администратор создает в базе данных итоговые таблицы. Затем это инструментальное средство анализировало запросы, прежде чем отправлять их на сервер Oracle. Если имеется итоговая таблица, позволяющая более эффективно ответить на запрос, Discoverer переписывает запрос так, чтобы он обращался к итоговым таблицам, а не к базовым, заданным в исходном запросе, после чего отправляет запрос серверу Oracle. Это было замечательно, пока для выполнения запросов использовалось именно это средство. Если аналогичный запрос выполнялся из среды SQL*Plus или поступал от клиента по протоколу JDBC, то перезапись запроса не происходила (не могла происходить). Более того, синхронизация между исходными и итоговыми данными не могла выполняться автоматически, поскольку это инструментальное средство не входило в состав сервера.

Начиная с версии 7.0, сервер Oracle уже поддерживал возможность, аналогичную итоговым таблицам, — *моментальные снимки* (snapshot). Первоначально эта возможность создавалась для поддержки репликации, но я лично использовал ее для сохранения ответов на большие запросы. Я создавал моментальные снимки, не использующие связь базы данных для репликации данных из одной базы в другую, а просто вычисляющие итоговые значения или выполняющие соединения для часто используемых данных. Это было здорово, но без возможности переписать запросы использование этого приема было ограниченным. Приложение должно было "знать" о существовании итоговых таблиц и использовать их, а это усложняло его создание и поддержку. При добавлении новой итоговой таблицы мне приходилось находить и переписывать код, в котором можно было ее использовать.

В версию Oracle 8.1.5 (Enterprise и Personal Edition) из инструментальных средств типа Discoverer была перенесена возможность переписывания запросов, предусмотрены механизмы автоматического обновления и планирования моментальных снимков (что позволило сделать итоговые таблицы "самоподдерживаемыми"), и все это объединено с воз-

можностью оптимизатора находить лучший план из многих альтернативных. В результате получились материализованные представления.

Все эти возможности, сосредоточенные в сервере, теперь позволяли **любому** приложению использовать преимущества автоматического переписывания запросов, независимо от способа доступа к базе данных: из утилиты SQL*Plus, из приложения Oracle Forms, через протоколы JDBC и ODBC, из программ Pro*C, OCI или из средства сторонних производителей. Любой сервер Oracle 8i масштаба предприятия позволяет управлять итоговыми таблицами. Кроме того, поскольку все происходит в базе данных, итоговые таблицы легко синхронизировать с исходными (по крайней мере сервер всегда "знает", когда они **не синхронизированы,** и может не использовать "устаревшие" итоговые таблицы (этим управляет пользователь)). Поскольку эти функциональные возможности включены непосредственно в сервер, любое приложение, способное обратиться к СУБД Oracle, может воспользоваться ими.

Тот же подход лежит в основе реализации средств тщательного контроля доступа (Fine Grained Access Control, FGAC, см. раздел об открытости в главе 1 и главу 21, посвященную этим средствам). Чем "ближе" к данным применяются функции, тем большее количество инструментальных средств сможет ими воспользоваться. Если поместить средства защиты вне базы данных, например в приложении, воспользоваться ими смогут только пользователи этого приложения (и то, если обращаются к данным исключительно через приложение).

Что необходимо для выполнения примеров

Для выполнения примеров, представленных в этой главе, необходима редакция Personal или Enterprise Edition сервера версии Oracle 8.1.5 и выше. Соответствующие функциональные возможности не поддерживаются в редакции Standard. Учетная запись, от имени которой будут выполняться примеры, должна иметь следующие привилегии:

- `GRANT CREATE SESSION`
- `GRANT CREATE TABLE`
- `GRANT CREATE MATERIALIZED VIEW`
- `GRANT QUERY REWRITE`

Первые три привилегии можно предоставить роли, которая, в свою очередь, предоставлена учетной записи. Привилегия **QUERY REWRITE** должна быть предоставлена непосредственно.

Кроме того, необходим доступ к табличному пространству, 30—50 Мбайт которого свободно.

Наконец, чтобы воспользоваться средствами переписывания запросов, необходимо использовать оптимизатор, основанный на стоимости (Cost-Based Optimizer — CBO). Если CBO не используется, запрос не может быть переписан. В наших примерах остав-

лена стандартная цель оптимизации, CHOOSE; чтобы воспользоваться возможностями переписывания запросов, достаточно проанализировать таблицы.

Пример

Продемонстрирую на простом примере, что может дать материализованное представление. Вы увидите, насколько сокращается время выполнения запроса за счет добавления итоговых данных. Запрос к большой таблице будет переписан сервером в запрос к гораздо меньшей таблице, причем без потери точности результата. Начнем с создания большой таблицы, содержащей список владельцев и принадлежащих им объектов. Таблица строится на основе представления ALL_OBJECTS словаря данных:

```
tkyte@TKYTE816> create table my_all_objects
  2  nologging
  3   as
  4  select * from all_objects
  5  union all
  6  select * from all_objects
  7  union all
  8  select * from all_objects
  9   /

Table created.

tkyte@TKYTE816> insert /*+ APPEND */ into my_all_objects
  2  select * from my_all_objects;

65742 rows created.

tkyte@TKYTE816> commit;

Commit complete.

tkyte@TKYTE816> insert /*+ APPEND */ into my_all_objects
  2  select * from my_all_objects;

131484 rows created.

tkyte@TKYTE816> commit;

Conmit complete.

tkyte@TKYTE816> analyze table my_all_objects compute statistics;

Table analyzed.
```

Моя система поддерживает язык Java (Java option), поэтому в таблице **MY_ALL_OBJECTS** после выполнения указанных действий оказалось около 250000 строк. Для получения такого же результата вам, возможно, придется выполнить **UNION ALL** и **INSERT** большее количество раз. Теперь выполним запрос к этой таблице, показывающий количество объектов у каждого пользователя. Первоначально для этого понадобится полный просмотр громадной таблицы:

```
tkyte@TKYTE816> set autotrace on
tkyte@TKYTE816> set timing on
tkyte@TKYTE816> select owner, count(*) from my_all_objects group by owner;
```

```
OWNER                          COUNT(*)

A                                    36
B                                    24
CTXSYS                             2220
DBSNMP                               48
DEMO                                 60
DEMO11                               36
DEMO_DDL                            108
MDSYS                              2112
MV_USER                              60
ORDPLUGINS                          312
ORDSYS                             2472
OUR_TYPES                            12
OUTLN                                60
PERFSTAT                            636
PUBLIC                           117972
SCHEDULER                            36
SCOTT                                84
SEAPARK                              36
SYS                              135648
SYSTEM                              624
TESTING                             276
TKYTE                                12
TTS_USER                             48
TYPES                                36

24 rows selected.

Elapsed: 00:00:03.35

tkyte@TKYTE816> set timing off
tkyte@TKYTE816> set autotrace traceonly
tkyte@TKYTE816> select owner, count(*) from my_all_objects group by owner;
24 rows selected.
```

Execution Plan

```
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2525 Card=24 Bytes=120)
   1    0   SORT (GROUP BY) <Cost=2525 Card=24 Bytes=120)
   2    1     TABLE ACCESS (FULL) OF 'MY_ALL_OBJECTS' (Cost=547 Card=262968
```

Statistics

```
          0  recursive calls
         27  db block gets
       3608  consistent gets
       3516  physical reads
          0  redo size
       1483  bytes sent via SQL*Net to client
        535  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          1  sorts (memory)
```

```
         0  sorts (disk)
        24  rows processed
```

Для получения результатов агрегирования необходимо просмотреть более 250000 записей в более чем 3600 блоках. К сожалению, в нашей системе этот запрос необходимо выполнять часто, по нескольку десятков раз в день. Приходится сканировать почти 30 Мбайт данных. Создав материализованное представление данных, можно избежать многократного подсчета по исходной таблице. Ниже описано, что для этого нужно сделать. Операторы GRANT и ALTER более детально будут рассмотрены в разделе "Как работают материализованные представления". Кроме указанных ниже привилегий может понадобиться также привилегия CREATE MATERIALIZED VIEW (в зависимости от того, какие роли предоставлены и действуют для соответствующей учетной записи):

```
tkyte@TKYTE816> grant query rewrite to tkyte;

Grant succeeded.

tkyte@TKYTE816> alter session set query_rewrite_enabled=true;

Session altered.

tkyte@TKYTE816> alter session set query_rewrite_integrity=enforced;

Session altered.

tkyte@TKYTE816> create materialized view my_all_objects_aggs
  2  build immediate
  3  refresh on commit
  4  enable query rewrite
  5  as
  6  select owner, count(*)
  7    from my_all_objects
  8   group by owner
  9  /

Materialized view created.

tkyte@TKYTE816> analyze table my_all_objects_aggs compute statistics;
Table analyzed.
```

По сути, мы заранее вычислили количество объектов и задали итоговую информацию в виде материализованного представления. Мы потребовали немедленно построить и наполнить данными это представление. Обратите внимание, что были также заданы конструкции **REFRESH ON COMMIT** и **ENABLE QUERY REWRITE** (вскоре они будут рассмотрены подробнее). Также обратите внимание, что, хотя создано материализованное представление, анализируется таблица. При создании материализованного представления создается настоящая таблица, и ее можно индексировать, анализировать и т.д.

Давайте посмотрим представление в действии, выполнив еще раз запрос, использовавшийся при создании представления:

```
tkyte@TKYTE816> set timing on
tkyte@TKYTE816> select owner, count(*)
  2    from my_all_objects
  3   group by owner;
```

```
OWNER                          COUNT(*)

A                                    36
B                                    24

TYPES                                36

24 rows selected.

Elapsed: 00:00:00.10

tkyte@TKYTE816> set timing off

tkyte6TKYTE816> set autotrace traceonly
tkyte@TKYTE816> select owner, count(*)
  2    from my_all_objects
  3   group by owner;

24 rows selected.

Execution Plan

   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=l Card=24 Bytes=216)
   1    0   TABLE ACCESS (FULL) OF 'MY_ALL_OBJECTS_AGGS' (Cost=l Card=Valve)

Statistics

          0  recursive calls
         12  db block gets
          7  consistent gets
          0  physical reads
          0  redo size
       1483  bytes sent via SQL*Net to client
        535  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         24  rows processed

tkyte@TKYTE816> set autotrace off
```

Вместо более чем 3600 **consistent gets** (логических операций ввода-вывода) использовано всего 12. Физического ввода-вывода на этот раз вообще не было — данные взяты из кэша. Теперь буферный кэш будет значительно эффективнее, так как кэшировать надо намного меньше данных. Раньше кэширование рабочего множества даже не начиналось, но теперь все рабочее множество помещается в кэше. Обратите внимание, что план выполнения запроса предусматривает полный просмотр таблицы **MY_ALL_OBJECTS_AGGS**, хотя запрос выполнялся к исходной таблице **MY_ALL_OBJECTS**. При получении запроса **SELECT OWNER, COUNT(*)**... сервер автоматически направляет его к соответствующему материализованному представлению.

Давайте пойдем дальше: добавим новую строку в таблицу **MY_ALL_OBJECTS** и зафиксируем изменение:

```
tkyte@TKYTE816> insert into my_all_objects
  2  (owner, object name, object type, object id)
```

```
  3  values
  4   ('New Owner', 'New Name', 'New Type', 1111111);

1 row created.

tkyte@TKYTE816> commit;

Commit complete.
```

Теперь выполним аналогичный запрос, но обратимся только к вновь вставленной строке:

```
tkyte@TKYTE816> set timing on
tkyte@TKYTE816> select owner, count(*)
  2    from my_all_objects
  3   where owner = 'New Owner'
  4   group by owner;

OWNER                          COUNT(*)

New Owner                             1

Elapsed: 00:00:00.01
tkyte@TKYTE816> set timing off

tkyte@TKYTE816> set autotrace traceonly
tkyte@TKYTE816> select owner, count(*)
  2    from my_all_objects
  3   where owner = 'New Owner'
  4   group by owner;

Execution Plan

   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=l Card=l Bytes=9)
   1    0   TABLE ACCESS (FULL) OF 'MY_ALL_OBJECTS_AGGS' (Cost=l Card=Valve)

Statistics

          0  recursive calls
         12  db block gets
          6  consistent gets
          0  physical reads
          0  redo size
        430  bytes sent via SQL*Net to client
        424  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed

tkyte@TKYTE816> set autotrace off
```

Анализ показывает, что новая строка была найдена при просмотре материализованного представления. Присутствие в исходном определении представления конструкции **REFRESH ON COMMIT** заставляет сервер Oracle обеспечивать синхронизацию между представлением и исходными данными — при изменении исходных данных изменяется

и представление. Такую синхронизацию нельзя обеспечить для всех материализованных представлений, но в случае однотабличного итогового представления (как наше) или только соединений, без агрегирования, это возможно.

Теперь еще один, последний запрос:

```
tkyte@TKYTE816> set timing on
tkyte@TKYTE816> select count(*)
  2    from my_all_objects
  3   where owner = 'New Owner';

  COUNT(*)

         1

Elapsed: 00:00:00.00

tkyte@TKYTE816> set timing off
tkyte@TKYTE816> set autotrace traceonly
tkyte@TKYTE816> select count(*)
  2    from my_all_objects
  3   where owner = 'New Owner';

Execution Plan

   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=9)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'MX_ALL_OBJECTS_AGGS' (Cost=1
-> Card=Valve)

Statistics

          0  recursive calls
         12  db block gets
          5  consistent gets
          0  physical reads
          0  redo size
        367  bytes sent via SQL*Net to client
        424  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed

tkyte@TKYTE816> set autotrace off
```

Как видите, сервер Oracle может использовать представление даже для запроса. Конструкции **GROUP BY** в нашем запросе не было, но сервер "понял", что материализованное представление все равно можно использовать. Это и является чудом использования материализованных представлений. Пользователи могут не знать о существовании итоговых таблиц, сервер сам разберется, что ответ уже существует, если включена воз-

можность переписывания запроса, и автоматически перепишет запрос так, чтобы использовать соответствующее материализованное представление. Это позволяет непосредственно повлиять на работу приложений, не изменяя в них ни одного запроса.

Назначение материализованных представлений

Его можно сформулировать коротко: **повышение производительности.** Получив (однажды) ответы на сложные вопросы, можно существенно снизить нагрузку на сервер. При этом:

- **Уменьшается количество физических чтений.** Приходится просматривать меньше данных.
- **Уменьшается количество записей.** Не нужно так часто сортировать/агрегировать данные.
- **Уменьшается нагрузка на процессор. Не** придется постоянно вычислять агрегаты и функции от данных, поскольку это уже сделано.
- **Существенно сокращается время ответа.** При использовании итоговых данных запросы выполняются значительно быстрее по сравнению с запросами к исходным данным. Все зависит от объема действий, которых можно избежать при использовании материализованного представления, но ускорение на несколько порядков вполне возможно.

При использовании материализованных представлений увеличивается потребность только в одном ресурсе — дисковом пространстве. Необходимо дополнительное место для хранения материализованных представлений, но за счет этого можно получить много преимуществ.

Материализованные представления больше подходят для сред, где данные только читаются (пусть даже интенсивно). Они **не** предназначены для использования в среде интенсивной обработки транзакций. Они требуют дополнительных затрат ресурсов при изменении базовых таблиц для учета этих изменений. При использовании опции **REFRESH ON COMMIT** возникают проблемы одновременного доступа. Вернемся к рассмотренному выше примеру с итоговыми данными. При вставке строки в базовую таблицу (или удалении) необходимо изменить одну из 24 строк в итоговой таблице, чтобы количество было актуальным. Это означает, что одновременно фиксировать транзакции сможет не более 24 пользователей (если, конечно, они затрагивают объекты разных владельцев). Однако это не мешает использовать материализованные представления в среде ООТ. Например, если периодически полностью обновлять представления (в периоды минимальной нагрузки), при изменении данных ресурсы дополнительно расходоваться не будут, как не будет и проблем с одновременным доступом. Это позволит создавать отчеты на основе, например, вчерашних данных, не обращаясь к активно изменяющимся при обработке транзакций данным.

Как работать с материализованными представлениями

Сначала работа с материализованными представлениями может показаться сложной. Иногда материализованное представление содержит ответ на определенный вопрос, но сервер Oracle почему-то его не использует. Если достаточно глубоко покопаться, можно понять, почему. Сервер Oracle — всего лишь программа, и она может работать только с явно предоставленной информацией. Чем больше *метаданных* предоставлено, чем больше сведений о базовых данных передано серверу Oracle, тем лучше. Эти фрагменты информации (требования **NOT NULL,** первичные ключи, внешние ключи и т.д.) — слишком тривиальные вещи, чтобы задумываться о них в среде хранилищ данных. Предоставляемые этими ключами и требованиями метаданные дают оптимизатору больше информации, а значит, — больше шансов.

Ключи и требования, подобные представленным выше, не только обеспечивают целостность данных, но и добавляют в словарь данных информацию о данных, которую можно использовать при переписывании запросов (отсюда и название — *метаданные).* Подробнее об этом см. в разделе "Требования целостности".

В следующих разделах мы рассмотрим, что необходимо сделать для использования материализованных представлений, представим ряд примеров и покажем, как после введения дополнительной информации, дополнительных метаданных в базу увеличивается частота использования материализованных представлений.

Подготовка

Для использования материализованных представлений обязательно надо установить один параметр инициализации, **COMPATIBLE.** Параметр **COMPATIBLE** должен иметь значение 8.1.0 или больше, чтобы переписывание запросов вообще применялось. Если этот параметр не будет иметь соответствующего значения, модуль переписывания запросов не будет вызываться.

Есть еще два связанных с использованием материализованных представлений параметра, которые можно устанавливать либо на уровне системы (в файле параметров инициализации, **INIT.ORA),** либо на уровне сеанса (с помощью оператора **ALTER SESSION).**

- **QUERY_REWRITE_ENABLED.** Если этот параметр не имеет значения **TRUE,** запроса не переписывается. Стандартное значение — **FALSE.**
- **QUERY_REWRITE_INTEGRITY.** Этот параметр управляет тем, **как** сервер Oracle переписывает запросы. Он может иметь одно из трех значений.
  - **ENFORCED.** Запросы будут переписываться с помощью требований и правил, применяемых и гарантируемых сервером Oracle. Имеются механизмы, с помощью которых можно сообщить серверу Oracle о других косвенных взаимосвязях, и это позволит переписать больше запросов; но поскольку сервер Oracle не обеспечивает эти взаимосвязи, он не будет использовать подобные сведения на этом уровне целостности.

- **TRUSTED**. Запросы будут переписываться на основе требований, обеспечиваемых сервером Oracle, а также всех взаимосвязей данных, о которых сообщили серверу, даже если их выполнение сервером не гарантируется. Например, в начальном примере можно вручную создать физическую таблицу **MY_ALL_OBJECTS_AGGS** с помощью распараллеливаемого и нерегистрируемого в журнале повторного выполнения оператора **CREATE TABLE AS SELECT** (для ускорения построения итоговой таблицы). Затем можно создать материализованное представление, использующее эту созданную заранее таблицу, а не создавать ее заново. Если необходимо, чтобы сервер Oracle использовал эту созданную заранее таблицу при последующем переписывании запросов, необходимо задать параметру **QUERY_REWRITE_INTEGRITY** значение **TRUSTED**. Надо, чтобы сервер Oracle "поверил", что мы предоставили корректные данные в заранее созданной таблице (сам сервер Oracle корректность этих данных не обеспечивает).
- **STALE_TOLERATED**. Запросы будут переписываться для использования материализованных представлений, даже если серверу Oracle известно, что содержащиеся в представлении данные устарели (не синхронизированы с исходными). Это может пригодиться в среде, где итоговые таблицы обновляются периодически, а не при фиксации изменений, и где небольшая рассинхронизация приемлема.

В представленном выше примере использовались операторы **ALTER SESSION**, обеспечивающие применение фокуса с переписыванием запроса. Поскольку в примере использовались только объекты и связи, поддерживаемые сервером Oracle, целостность запроса при перезаписи можно установить максимальной: **ENFORCED**.

Также необходимо получить привилегию **QUERY REWRITE**. Но учетной записи, от имени которой я работаю, предоставлена роль **DBA**, среди привилегий которой есть и **QUERY REWRITE**, так зачем же явно предоставлять эту привилегию самому себе? Причина в том, что нельзя создать скомпилированные хранимые объекты, будь то материализованные представления, хранимые процедуры или триггеры, имея привилегии роли (роли **DBA** в данном случае). Полное описание особенностей использования ролей при работе со скомпилированными хранимыми объектами дано в главе 23. Если создается материализованное представление при включенном параметре **QUERY_REWRITE_ENABLED**, но системная привилегия **QUERY REWRITE** явно не предоставлена, будет получено следующее сообщение об ошибке:

```
create materialized view my_all_objects_aggs

ERROR at line 1:
ORA-01031: insufficient privileges
```

Внутренняя реализация

Итак, теперь, научившись создавать материализованные представления и убедившись, что они используются, разберемся, что будет предпринимать сервер Oracle для переписывания запросов? Обычно, когда параметр **QUERY_REWRITE_ENABLED** имеет значение **FALSE**, сервер Oracle анализирует полученный оператор SQL и оптимизирует его.

При включенном переписывании запросов сервер Oracle добавляет в этот процесс дополнительный шаг. После анализа сервер попытается переписать запрос так, чтобы он обращался к тому или иному материализованному представлению вместо указанной в нем таблицы. Если переписать запрос можно, полученный в результате запрос (или запросы) анализируется и оптимизируется вместе с исходным запросом. Из полученного набора выбирается план выполнения с наименьшей стоимостью. Если запрос переписать не удается, исходный проанализированный запрос просто оптимизируется и выполняется, как обычно.

Переписывание запроса

При включенном переписывании запроса сервер Oracle будет пытаться переписать запрос так, чтобы он обращался к материализованному представлению, в следующих случаях.

Полное совпадение текста

Сервер ищет полное совпадение строк запроса с текстами определяющих запросов, хранящихся в словаре данных для материализованных представлений. В рассмотренном ранее примере именно этот метод использовал сервер Oracle для первого запроса, при выполнении которого использовалось материализованное представление. При этом используется более "дружественный" (гибкий) алгоритм, чем при поиске в разделяемом пуле (требующем побайтового совпадения), поскольку пробелы, регистр символов и другие особенности форматирования игнорируются.

Частичное совпадение текста

Начиная с конструкции **FROM,** оптимизатор сравнивает оставшийся текст с текстом запроса, определяющего материализованное представление. В результате допускаются расхождения в списке выбора. Если необходимые данные можно получить из материализованного представления (т.е. по нему можно найти значения всех выражений, указанных в списке выбора), сервер Oracle перепишет запрос с использованием этого материализованного представления. Запрос **SELECT LOWER(OWNER) FROM MY_ALL_OBJECTS GROUP BY OWNER;** — пример частичного совпадения текста.

Общие методы переписывания запроса

Они обеспечивают использование материализованного представления, даже если оно содержит часть необходимых данных, больше данных или данные, которые могут быть преобразованы к нужному виду. Оптимизатор сравнивает определение материализованного представления с отдельными компонентами запроса **(SELECT, FROM, WHERE, GROUP BY)** в поисках соответствия. При этом сервер Oracle проверяет для этих компонентов следующее.

- **Достаточность данных.** Можно ли получить нужные данные из этого материализованного представления? Если в списке выбора есть столбец X, отсутствующий в материализованном представлении, и его нельзя получить, выполняя соединение с этим представлением, то сервер Oracle не будет переписывать запрос так, чтобы он обращался к этому представлению. Например, запрос **SELECT**

DISTINCT OWNER FROM MY_ALL_OBJECTS при наличии созданного в примере материализованного представления может быть переписан, поскольку столбец **OWNER** доступен. Запрос же **SELECT DISTINCT OBJECT_TYPE FROM MY_ALL_OBJECTS** по материализованному представлению выполнить нельзя, поскольку в нем недостаточно данных.

- **Совместимость по соединениям.** Можно ли получить результат соединения, требуемого исходным запросом, из материализованного представления.

Совместимость по соединениям можно продемонстрировать с помощью таблицы **MY_ALL_OBJECTS** и следующих таблиц:

```
tkyte@TKYTE816> create table t1 (owner varchar2(30), flag chard));

Table created.

tkyte@TKYTE816> create table t2 (object_type varchar2(30), flag char(1));

Table created.
```

Следующий запрос совместим по соединению с материализованным представлением, поэтому он будет переписан так, чтобы обращаться к материализованному представлению:

```
tkyte@TKYTE816> select a.owner, count(*), b.owner
  2    from my_all_objects a, t1 b
  3   where a.owner = b.owner
  4     and b.flag is not null
  5   group by a.owner, b.owner
  6   /
```

Сервер может выяснить, что при использовании материализованного представления вместо исходной таблицы будет получен тот же ответ. Следующий запрос, однако, хотя и похож, но не совместим по соединению:

```
tkyte@TKYTE816> select a.owner, count(*), b.object_type
  2    from my_all_objects a, t2 b
  3   where a.object_type = b.object_type
  4     and b.flag is not null
  5   group by a.owner, b.object_type
  6   /
```

Столбец **OBJECT_TYPE** в наше материализованное представление не входит, поэтому сервер Oracle не может переписать запрос так, чтобы он обращался к этому представлению.

Совместимость конструкций группировки

Она требуется, если и материализованное представление, и запрос содержат конструкцию GROUP BY. Если материализованное представление сгруппировано на необходимом или более высоком уровне детализации, запрос будет переписан с использованием материализованного представления. Запрос **SELECT COUNT(*) FROM MY_ALL_OBJECTS GROUP BY 1**, если выполнять его после представленного выше примера, представляет тот самый случай, когда материализованное представление сгруп-

пировано на более высоком уровне детализации, чем необходимо. Сервер может переписать этот запрос с использованием материализованного представления, хотя критерии группировки в запросе и материализованном представлении не совпадают.

Совместимость конструкций агрегирования

Такая совместимость требуется, если и запрос, и материализованное представление содержат функции агрегирования. Она гарантирует, что материализованное представление обеспечит данные для необходимых агрегатов. В некоторых случаях возможны очень интересные варианты переписывания. Например, сервер распознает, что **AVG(X)** — то же самое, что и **SUM(X)/COUNT(X),** так что запрос, требующий выбора **AVG(X)**, может быть выполнен по материализованному представлению, содержащему значения **SUM** и **COUNT**.

Во многих случаях простое применение описанных выше правил позволит серверу Oracle переписать запрос так, чтобы он обращался к материализованному представлению. В других случаях (как будет показано далее) серверу потребуется помощь администратора. Надо предоставить серверу дополнительную информацию, чтобы он смог использовать для ответа на запрос материализованное представление.

Как гарантировать использование представлений

В этом разделе мы научимся это делать — сначала с помощью требований, помогающих использовать переписывание запроса, а потом с помощью *измерений* (dimensions), являющихся средством описания сложных взаимосвязей — иерархий данных.

Требования целостности

Меня часто спрашивали: "Почему надо использовать первичный ключ? Почему просто не создать уникальный индекс?". Можно, конечно, создать и просто индекс, но ведь факт использования первичного ключа говорит намного больше, чем просто создание уникального индекса. То же самое можно сказать об использовании внешних ключей, требований **NOT NULL** и других. Они не только защищают данные от нежелательных изменений, но и добавляют информацию о данных в словарь данных. На основе этой дополнительной информации сервер Oracle сможет чаще и в более сложных случаях переписывать запрос.

Рассмотрим следующий небольшой пример. Скопируем таблицы **EMP** и **DEPT** из схемы пользователя **SCOTT** и создадим материализованное представление, соединяющее эти две таблицы. Это материализованное представление отличается от использованного в первом примере тем, что для него задана конструкция **REFRESH ON DEMAND.** Это означает, что для учета изменений в исходных данных это представление надо обновлять вручную:

```
tkyte@TKYTE816> create table emp as select * from scott.emp;

Table created.

tkyte@TKYTE816> create table dept as select * from scott.dept;
```

```
Table created.

tkyte@TKYTE816> alter session set query_rewrite_enabled=true;

Session altered.

tkyte@TKYTE816> alter session set query_rewrite_integrity=enforced;

Session altered.

tkyte@TKYTE816> create materialized view emp_dept
  2  build immediate
  3  refresh on demand
  4  enable query rewrite
  5  as
  6  select dept.deptno, dept.dname, count (*)
  7    from emp, dept
  8   where emp.deptno = dept.deptno
  9   group by dept.deptno, dept.dname
 10  /

Materialized view created.

tkyte@TKYTE816> alter session set optimizer_goal=all_rows;

Session altered.
```

Поскольку базовые таблицы и полученное материализованное представление — очень небольшие, мы с помощью оператора ALTER SESSION принудительно потребуем использовать оптимизатор, основанный на стоимости, а не проанализируем таблицы, как обычно. Если сервер Oracle "узнает", насколько малы эти таблицы, он не будет выполнять некоторые из желательных для нас оптимизаций. При использовании стандартной статистической информации, оптимизатор будет работать так, будто таблицы достаточно большие.

В данном случае мы предоставили серверу Oracle мало информации. Ему не известно, как соотносятся таблицы EMP и DEPT, какие столбцы являются первичными ключами и т.д. Теперь выполним запрос и посмотрим, что произойдет:

```
tkyte@TKYTE816> set autotrace on
tkyte@TKYTE816> select count(*) from emp;

  COUNT(*)

        14

Execution Plan

   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1 Card=1)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'EMP' (Cost=1 Card=82)
```

Запрос был выполнен к базовой таблице **EMP**. Теперь мы с вами знаем, что значение **COUNT(*)** намного эффективнее (особенно при большом количестве отделов и сотрудников в них) может быть получено из материализованного представления. В нем есть вся необходимая информация для подсчета количества сотрудников. Мы знаем об этом потому, что учитываем сведения о данных, неизвестные серверу Oracle:

- Столбец **DEPTNO** — первичный ключ таблицы **DEPT.** Это означает, что каждая строка в таблице **EMP** соответствует не более чем одной строке в таблице **DEPT.**
- Столбец **DEPTNO** в таблице **EMP** — внешний ключ по столбцу **DEPTNO** таблицы **DEPT.** Если значение столбца **DEPTNO** в строке таблицы **EMP** непустое, она будет соединена со строкой в таблице **DEPT** (ни одна строка таблицы **EMP** с непустым значением при соединении потеряна не будет).
- Для столбца **DEPTNO** в таблице **EMP** задано требование **NOT NULL. В** сочетании с требованием внешнего ключа это означает, что **ни одна** строка таблицы **EMP** не будет потеряна.

Эти три факта в совокупности означают, что при соединении таблиц **EMP** и **DEPT** каждая строка таблицы **EMP** будет входить в результирующее множество **только один раз.** Поскольку серверу Oracle об этом не сообщалось, он не смог использовать материализованное представление. Давайте же сообщим серверу все это:

```
tkyte@TKYTE816> alter table dept
  2 add constraint dept_pk primary key(deptno);

Table altered.

tkyte@TKYTE816> alter table emp
  2  add constraint emp_fk_dept
  3 foreign key(deptno) references dept(deptno);

Table altered.

tkyte@TKYTE816> alter table emp modify deptno not null;

Table altered.
tkyte@TKYTE816> set autotrace on
tkyte@TKYTE816> select count(*) from emp;
  COUNT(*)

        14

Execution Plan

   0     SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1 Card=1 Bytes=13)
   1    0   SORT (AGGREGATE)
   2    1    TABLE ACCESS (FULL) OF 'EMP DEPT' (Cost=1 Card=82 Bytes=1066)
```

Теперь сервер Oracle может переписать запрос с использованием материализованного представления **EMP_DEPT.** Каждый раз, когда известно, что сервер **мог бы** использовать материализованное представление, но **не использует (и** проверено, что вообще использование материализованных представлений в сеансе возможно), более детально изучите данные и спросите себя: "Какую информацию я не предоставил серверу Oracle?". В девяти случаях из десяти обнаружится фрагмент метаданных, при добавлении которого сервер Oracle будет переписывать запрос.

Итак, что же произойдет в реальном хранилище данных, где в представленных таблицах будут десятки миллионов записей? Дополнительные затраты ресурсов на проверку выполнения требования целостности нежелательны — в программе первичной обра-

ботки данных это уже сделано, не так ли? В данном случае можно создать непроверяемое требование, которое используется для информирования сервера о взаимосвязи, но сервером не проверяется. Давайте рассмотрим предыдущий пример еще раз, но теперь сымитируем загрузку данных в существующее хранилище (хранилище представлено предыдущим примером). Удалим требования, загрузим данные, обновим материализованные представления и снова добавим требования. Начнем с удаления требований:

```
tkyte@TKYTE816> alter table emp drop constraint emp_fk_dept;

Table altered.

tkyte@TKYTE816> alter table dept drop constraint dept_pk;

Table altered.

tkyte@TKYTE816> alter table emp modify deptno null;

Table altered.
```

Теперь, чтобы сымитировать загрузку, я вставлю новую строку (для демонстрационных целей этого вполне достаточно) в таблицу EMP. Затем мы обновим материализованное представление и сообщим серверу Oracle, что его можно считать актуальным (FRESH):

```
tkyte@TKYTE816> insert into emp (empno,deptno) values (1, 1);

1 row created.

tkyte@TKYTE816> exec dbms_mview.refresh('EMP_DEPT');

PL/SQL procedure successfully completed.

tkyte@TKYTE816> alter materialized view emp_dept consider fresh;

Materialized view altered.
```

Теперь сообщаем серверу о взаимосвязи таблиц **EMP** и **DEPT**:

```
tkyte@TKYTE816> alter table dept
  2  add constraint dept_pk primary key(deptno)
  3  rely enable NOVALIDATE
  4  /

Table altered.

tkyte@TKYTE816> alter table emp
  2  add constraint emp_fk_dept
  3  foreign key(deptno) references dept(deptno)
  4  rely enable NOVALIDATE
  5  /

Table altered.

tkyte@TKYTE816> alter table emp modify deptno not null NOVALIDATE;

Table altered.
```

Итак, мы сообщили серверу Oracle, что имеется, как и прежде, внешний ключ в таблице **EMP**, ссылающийся на таблицу **DEPT**. Однако, поскольку перед загрузкой в хранилище данные уже обрабатывались, мы сообщаем серверу, что проверять выполнение требований не надо. Опция **NOVALIDATE** позволяет избежать проверки загруженных

данных, а опция RELY требует, чтобы сервер рассматривал данные как целостные. По сути, мы сообщили серверу о необходимости считать, что при соединении таблиц EMP и DEPT по столбцу DEPTNO каждая строка в таблице EMP обязательно попадет в результат, причем не более одного раза.

Фактически мы "обманули" сервер, вставив в таблицу EMP строку, для которой нет соответствующей строки в таблице DEPT. Теперь все готово для выполнения запроса:

```
tkyte@TKYTE816> alter session set query_rewrite_integrity=enforced;
Session altered.
tkyte@TKYTE816> select count(*) from emp;

  COUNT(*)

        15

Execution Plan

   0     SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1 Card=1)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'EMP' (Cost=1 Card=164)
```

Поскольку установлено значение параметра **QUERY_REWRITE_INTEGRITY=ENFORCED**, сервер Oracle не переписал запрос с использованием материализованного представления. Необходимо понизить уровень целостности запроса. Надо, чтобы сервер Oracle нам "поверил":

```
tkyte@TKYTE816> alter session set query_rewrite_integrity=trusted;
Session altered.
tkyte@TKYTE816> select count(*) from emp;

  COUNT(*)

        14

Execution Plan

   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1 Card=1 Bytes=13)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'EMP_DEPT' (Cost=1 Card=82 Bytes=1066)
```

В этом случае сервер Oracle переписал запрос, но побочным эффектом оказалось то, что вновь вставленные строки не учтены. Возвращается "ошибочный" ответ, поскольку "факт" сохранения каждой строки таблицы **EMP** в результатах соединения с таблицей DEPT при загруженных в таблицу данных — уже не факт. При обновлении материализованного представления вновь добавленная строка **EMP** в него не попала. Данные, которым сервер Oracle по нашему требованию доверял, оказались ненадежными. В результате мы приходим к двум важным умозаключениям:

- можно очень эффективно использовать материализованные представления в больших хранилищах данных, без необходимости выполнять множество дополнительных и зачастую избыточных проверок данных;

- **но, лучше лишний раз перепроверить согласованность данных, если вы требуете от сервера Oracle им доверять.**

Измерения

Использование измерений — еще один метод предоставления дополнительной информации серверу Oracle. Предположим, имеется таблица исходных данных с датами транзакций и идентификаторами клиентов. По дате транзакции в другой таблице можно найти детальную информацию о том, к какому месяцу относится транзакция, к какому кварталу финансового года и т.д. Теперь предположим, что создано материализованное представление для хранения агрегированной информации о продажах по месяцам. Может ли сервер Oracle использовать это представление, выполняя запрос о продажах за квартал или год? Да, мы знаем, что по дате транзакции можно получить месяц, по месяцу — квартал, по кварталу — год, так что — да, **может.** Серверу Oracle (пока) об этой взаимосвязи не известно, поэтому использовать представление он не будет.

С помощью объекта базы данных **DIMENSION** *(измерение)* можно сообщить серверу Oracle эти сведения о данных, чтобы он использовал для переписывания большего количества запросов. Измерение декларирует отношение главный/подчиненный между парами столбцов. С его помощью можно указать серверу Oracle что, в строке таблицы значение столбца **MONTH** определяет значение, которое окажется в столбце **QTR,** столбец **QTR** определяет значение, которое окажется в столбце **YEAR** и т.д. Используя измерение, можно создать материализованное представление, содержащее менее подробные сведения, чем исходные записи (например, итоговые данные по месяцам). Этот уровень агрегирования может оказаться более детальным, чем требуется в запросе (в запросе, скажем, требуются данные по кварталам), но сервер Oracle разберется, что для получения ответа можно использовать материализованное представление.

Вот простой пример. Создадим таблицу **SALES** для хранения даты транзакции, идентификатора клиента и общей суммы продаж. В этой таблице будет около 350000 строк. Другая таблица, **TIME_HIERARCHY,** будет содержать соответствие даты транзакции месяцу, кварталу и году. При соединении этих двух таблиц можно получить агрегированные данные по месяцам, кварталам, годам и т.д. Аналогично, если имеется таблица, сопоставляющая идентификатор клиента с почтовым индексом, а почтовые индексы — с регионом, можно легко соединить эту таблицу с таблицей **SALES** для агрегирования данных по почтовому индексу или региону.

В обычной базе данных (без материализованных представлений и других специфических структур) эти действия можно выполнить, но это потребует много времени. Для каждой строки данных продаж придется выполнять чтение по индексу справочной таблицы (соединение вложенным циклом **NESTED LOOP JOIN)** для преобразования даты транзакции или идентификатора клиента в другое значение и последующего группирования результатов по этому значению. Вот тут и пригодится материализованное представление. Можно хранить итоговые данные по продажам, агрегированные, скажем, помесячно по датам транзакции и по почтовым индексам клиентов. Теперь обобщение данных поквартально или по регионам может выполняться очень быстро.

Начнем с создания таблицы **SALES** и загрузки в нее случайных тестовых данных, сгенерированных на основе представления **ALL_OBJECTS.**

```
tkyte@TKYTE816> create table sales
  2  (trans_date date, cust_id int, sales_amount number);

Table created.

tkyte@TKYTE816> insert /*+ APPEND */ into sales
  2  select trunc(sysdate,'year')+mod(rownum,366) TRANS_DATE,
  3       mod(rownum,100)        CUST_ID,
  4       abs(dbms_random.random)/100        SALES_AMOUNT
  5     from all_objects
  6  /

21921 rows created.

tkyte@TKYTE816> commit;

Commit complete.
```

Эта исходная информация будет представлять данные за год. Я задаю столбец **TRANS_DATE** как первый день года плюс число от 1 до 365. Значение **CUST_ID** — число от 0 до 99. Общая сумма продаж — некоторое сравнительно большое число (год выдался хороший).

В моем представлении **ALL_OBJECTS** содержится около 22000 строк, так что после четырех вставок, каждая из которых удваивает размер таблицы, мы получим около 350000 записей. Я использую подсказку **/*+ APPEND */**. чтобы избежать генерации большого объема данных в журнал повторного выполнения:

```
tkyte@TKYTE816> begin
  2      for i in 1 .. 4
  3      loop
  4          insert /*+ APPEND */ into sales
  5          select trans_date, cust_id, abs(dbms_random.random)/100
  6            from sales;
  7          commit;
  8      end loop;
  9  end;
 10  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select count(*) from sales;

  COUNT(*)

    350736
```

Теперь необходимо создать таблицу **TIME_HIERARCHY**, "округляющую" дату до месяца, года, квартала и т.д.:

```
tkyte@TKYTE816> create table time_hierarchy
  2  (day primary key, mmyyyy, mon_yyyy, qtr_yyyy, yyyy)
  3  organization index
  4  as
  5  select distinct
  6     trans_date    DAY,
  7     cast (to_char(trans_date,'mmyyyy') as number) MMYYYY,
```

```
  8     to_char(trans_date,'mon-yyyy') MON_YYYY,
  9     'Q' || ceil( to_char(trans_date,'mm')/3) ||  'FY'
 10     ||     to_char(trans_date,'yyyy') QTR_YYYY,
 11    cast( to_char( trans_date,  'yyyy') as number ) YYYY
 12    from sales
 13  /

Table created.
```

В данном случае все просто. Мы сгенерировали столбцы:

- **MMYYYY** — месяц и год;
- **MON_YYYY** — то же, но с сокращенным названием месяца;
- **QTR_YYYY** — квартал и год;
- **YYYY** - год.

Однако вычисления, необходимые для создания подобной таблицы, могут быть намного сложнее. Например, кварталы финансового года вычислить не так легко, как и границы финансового года. Как правило, его границы не соответствуют календарному году.

Теперь создадим материализованное представление **SALES_MV**. Оно суммирует исходные продажи за месяц. Можно ожидать, что в полученном материализованном представлении будет примерно 1/30 общего количества строк таблицы **SALES,** если данные были равномерно распределены:

```
tkyte@TKYTE816> analyze table sales compute statistics;

Table analyzed.

tkyte@TKYTE816> analyze table time_hierarchy confute statistics;

Table analyzed.

tkyte@TKYTE816> create materialized view sales_mv
  2  build immediate
  3  refresh on demand
  4  enable query rewrite
  5  as
  6  select sales.cust_id, sum(sales.sales_amount) sales_amount,
  7         time_hierarchy.mmyyyy
  8    from sales, time_hierarchy
  9   where sales.trans_date = time_hierarchy.day
 10   group by sales.cust_id, time_hierarchy.mmyyyy
 11  /

Materialized view created.

tkyte@TKYTE816> set autotrace on
tkyte@TKYTE816> select time_hierarchy.mmyyyy, sum(sales_amount)
  2    from sales, time_hierarchy
  3   where sales.trans_date = time_hierarchy.day
  4   group by time_hierarchy.mmyyyy
  5  /
```

```
MMYYY SUM(SALES_AMOUNT)

 12001        3.2177E+11
 12002        1.0200E+10
 22001        2.8848E+11
 32001        3.1944E+11
 42001        3.1012E+11
 52001        3.2066E+11
 62001        3.0794E+11
 72001        3.1796E+11
 82001        3.2176E+11
 92001        3.0859E+11
102001        3.1868E+11
112001        3.0763E+11
122001        3.1305E+11

13 rows selected.

Execution Plan

  0      SELECT STATEMENT Optimizer=CHOOSE (Cost=4 Card=327 Bytes=850VALVE)
  1    0   SORT (GROUP BY) (Cost=4 Card=327 Bytes=8502)
  2    1     TABLE ACCESS (FULL) OF 'SALES_MV' (Cost=1 Card=327 Bytes
```

Пока все отлично: сервер Oracle переписал запрос так, что используется представление **SALES_MV**. Однако посмотрим, что произойдет при выполнении запроса, требующего более высокого уровня агрегирования:

```
tkyte@TKYTE816> set timing on
tkyte@TKYTE816> set autotrace on
tkyte@TKYTE816> select time_hierarchy.qtr_yyyy, sum(sales_amount)
  2   from sales, time_hierarchy
  3  where sales.trans_date = time_hierarchy.day
  4  group by time_hierarchy.qtr_yyyy
  5  /

QTR_YYYY                                    SUM(SALES_AMOUNT)

Q1 FY2001                                        9.2969E+11
Q1 FY2002                                        1.0200E+10
Q2 FY2001                                        9.3872E+11
Q3 FY2001                                        9.4832E+11
Q4 FY2001                                        9.3936E+11

Elapsed: 00:00:05.58

Execution Plan

  0      SELECT STATEMENT Optimizer=CHOOSE (Cost=8289 Card=5 Bytes=14)
  1    0   SORT (GROUP BY) (Cost=8289 Card=5 Bytes=145)
  2    1     NESTED LOOPS (Cost=169 Card=350736 Bytes=10171344)
  3    2       TABLE ACCESS (FULL) OF 'SALES' (Cost=169 Card=350736 B
  4    2       INDEX (UNIQUE SCAN) OF 'SYS IOT TOP 30180' (UNIQUE)
```

```
Statistics

          0  recursive calls
         15  db block gets
     351853  consistent gets
```

Как видите, сервер Oracle не знает того, что знаем мы. Он еще не знает, что мог бы использовать материализованное представление для ответа на данный запрос, поэтому использует исходную таблицу **SALES**, проделывая огромный объем работы для получения ответа. То же самое получится и при запросе обобщенных данных за финансовый год.

С помощью объекта **DIMENSION** проинформируем сервер Oracle о том, что материализованное представление позволяет получить ответ и на эти запросы. Сначала создадим объект **DIMENSION**:

```
tkyte@TKYTE816>create dimension time_hierarchy_dim
  2         level day      is time_hierarchy.day
  3         level mmyyyy   is time_hierarchy.mmyyyy
  4         level qtr_yyyy is time_hierarchy.qtr_yyyy
  5         level yyyy     is time_hierarchy.yyyy
  6  hierarchy time_rollup
  7   (
  8    day child of
  9    mmyyyy child of
 10    qtr_yyyy child of
 11    yyyy
 12   )
 13   attribute mmyyyy
 14   determines mon_yyyy;

Dimension created.
```

Этот оператор сообщает серверу Oracle, что столбец **DAY** таблицы **TIME_HIERARCHY** определяет значение столбца **MMYYYY**, который, в свою очередь, определяет значение столбца **QTR_YYYY**. Наконец, значение столбца **QTR_YYYY** определяет значение столбца **YYYY**. Также утверждается, что столбцы **MMYYYY** и **MON_YYYY** — синонимы, между ними есть однозначное соответствие. Так что, когда сервер Oracle обнаруживает в запросе столбец **MON_YYYY**, он обрабатывает запрос так же, как при использовании столбца **MMYYYY**. Теперь, когда серверу Oracle известна взаимосвязь данных, выполнение запроса существенно ускоряется:

```
tkyte@TKYTE816>set autotrace on
tkyte@TKYTE816> select time_hierarchy.qtr_yyyy, sum(sales_amount)
  2   from sales, time_hierarchy
  3  where sales.trans_date = time_hierarchy.day
  4  group by time_hierarchy.qtr_yyyy
  5   /

QTR_YYYY                                SUM(SALES_AMOUNT)

Q1 FY2001                                     9.2969E+11
Q1 FY2002                                     1.0200E+10
```

```
Q2 FY2001                                                    9.3872E+11
Q3 FY2001                                                    9.4832E+11
Q4 FY2001                                                    9.3936E+11

Elapsed: 00:00:00.20

Execution Plan

   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=7 Card=5 Bytes=195)
   1    0   SORT (GROUP BY) (Cost=7 Card=5 Bytes=195)
   2    1     HASH JOIN (Cost=6 Card=150 Bytes=5850)
   3    2       VIEW (Cost=4 Card=46 Bytes=598)
   4    3         SORT (UNIQUE) (Cost=4 Card=46 Bytes=598)
   5    4           INDEX (FAST FULL SCAN) OF 'SYS_IOT_TOP_30180' (UNI
   6    2       TABLE ACCESS (FOIL) OF 'SALES_MV (Cost=1 Card=327 Byt

Statistics

          0  recursive calls
         16  db block gets
         12  consistent gets
```

Мы сократили количество логических чтений с 350000 до 12 — не так уж плохо. Если выполнить этот пример, различие будет заметно. Для выполнения первого запроса потребовалось некоторое время (около шести секунд), а вот ответ на второй оказался на экране раньше, чем я отпустил клавишу Enter (через две сотых доли секунды).

Для одной базовой исходной таблицы можно задавать сколько угодно иерархий с помощью **DIMENSION**. Давайте свяжем с каждым клиентом в таблице продаж атрибуты **ZIP_CODE** (почтовый индекс) и **REGION** (регион):

```
tkyte@TKYTE816> create table customer_hierarchy
  2  (cust_id primary key, zip_code, region)
  3  organization index
  4  as
  5  select cust_id,
  6    mod(rownum, 6) || to_char(mod( rownum, 1000 ), 'fm0000') zip_code,
  7    mod(rownum, 6) region
  8    from (select distinct cust_id from sales)
  9  /

Table created.

tkyte@TKYTE816> analyze table customer_hierarchy compute statistics;

Table analyzed.
```

Теперь пересоздадим материализованное представление так, чтобы значения **SALES_AMOUNT** группировались по столбцам **ZIP_CODE** и **MMYYYY**:

```
tkyte@TKYTE816> drop materialized view sales_mv;

Materialized view dropped.

tkyte@TKYTE816> create materialized view sales_mv
  2  build immediate
  3  refresh on demand
```

```
  4  enable query rewrite
  5  as
  6  select customer_hierarchy.zip_code,
  7         time_hierarchy.mnyyyy,
  8         sum(sales.sales_amount)  sales_amount
  9    from sales, time_hierarchy, customer_hierarchy
 10   where sales.trans_date = time_hierarchy.day
 11     and sales.cust_id = customer_hierarchy.cust_id
 12   group by customer_hierarchy.zip_code, time_hierarchy.mmyyyy
 13  /

Materialized view created.
```

Выполнив запрос, который выдает данные по продажам, сгруппированные по столбцам ZIP_CODE и MMYYYY, можно убедиться, что для их выполнения используется это материализованное представление:

```
tkyte@TKYTE816> set autotrace
tkyte@TKYTE816> select customer_hierarchy.zip_code,
  2         time_hierarchy.mmyyyy,
  3         sum(sales.sales_amount)  sales_amount
  4    from sales, time_hierarchy, customer_hierarchy
  5   where sales.trans_date = tiroe_hierarchy.day
  6     and sales.cust_id = customer_hierarchy.cust_id
  7   group by customer_hierarchy.zip_code, time_hierarchy.mmyyyy
  8  /

1250 rows selected.

Execution Plan

   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=409 Bytes=204
   1    0   TABLE ACCESS (FULL) OF 'SALES_MV' (Cost=1 Card=409 Bytes=2

Statistics

         28  recursive calls
         12  db block gets
        120  consistent gets
```

Однако если запросить информацию на другом уровне агрегирования (обобщая **MMYYYY** до **YYYY** и **ZIP_CODE** до **REGION**), окажется, что сервер не счел возможным использовать материализованное представление:

```
tkyte@TKYTE816> select customer_hierarchy.region,
  2         time_hierarchy.yyyy,
  3         sum(sales.sales_amount)  sales_amount
  4    from sales, time_hierarchy, customer_hierarchy
  5   where sales.trans_date = time_hierarchy.day
  6     and sales.cust_id = customer_hierarchy.cust_id
  7   group by customer_hierarchy.region, time_hierarchy.yyyy
  8  /
9 rows selected.
```

```
Execution Plan

   0       SELECT STATEMENT Optimizer=CHOOSE (Cost=8289 Card=9 Bytes=26
   1    0    SORT (GROUP BY) (Cost=8289 Card=9 Bytes=261)
   2    1      NESTED LOOPS (Cost=169 Card=350736 Bytes=10171344)
   3    2        NESTED LOOPS (Cost=169 Card=350736 Bytes=6663984)
   4    3          TABLE ACCESS (FULL) OF 'SALES' (Cost=169 Card=350736
   5    3          INDEX (UNIQUE SCAN) OF 'SYS_IOT_TOP_30185' (UNIQUE)
   6    2        INDEX (UNIQUE SCAN) OF 'SYS_IOT_TOP_30180' (UNIQUE)

Statistics

          0  recursive calls
         15  db block gets
     702589  consistent gets
```

Сервер учел имеющееся измерение по времени, но у него отсутствует информация о том, как соотносятся столбцы **CUST_ID, ZIP_CODE** и **REGION** в таблице **CUSTOMER_HIERARCHY**. Чтобы исправить это, пересоздадим измерение так, чтобы оно включало две иерархии: одну для таблицы **TIME_HIERARCHY**, а другую — для **CUSTOMER_HIERARCHY**:

```
tkyte@TKYTE816> drop dimension time_hierarchy_dim
  2  /

Dimension dropped.

tkyte@TKYTE816> create dimension sales_dimension
  2       level cust_id   is customer_hierarchy.cust_id
  3       level zip_code  is customer_hierarchy.zip_code
  4       level region    is customer_hierarchy.region
  5       level day       is time_hierarchy.day
  6       level mmyyyy    is time_hierarchy.mmyyyy
  7       level qtr_yyyy  is time_hierarchy.qtr_yyyy
  8       level yyyy      is time_hierarchy.yyyy
  9  hierarchy cust_rollup
 10  (
 11       cust_id child of
 12       zip_code child of
 13       region
 14  )
 15  hierarchy time_rollup
 16  (
 17       day child of
 18       mmyyyy child of
 19       qtr_yyyy child of
 20       yyyy
 21  )
 22  attribute mmyyyy
 23  determines mon_yyyy;

Dimension created.
```

Мы удалили исходную иерархию по времени и создали новое, более информативное измерение, описывающее все существенные взаимосвязи. Теперь сервер Oracle "поймет", что по созданному представлению **SALES_MV** можно ответить на многие другие запросы. Например, если еще раз запросить "регионы по годам":

```
tkyte@TKYTE816> select customer_hierarchy.region,
  2         time_hierarchy.yyyy,
  3         sum(sales.sales_amount)  sales_amount
  4    from sales, time_hierarchy, customer_hierarchy
  5   where sales.trans_date = time_hierarchy.day
  6     and sales.cust_id = customer_hierarchy.cust_id
  7   group by customer_hierarchy.region, time_hierarchy.yyyy
  8  /

REGION       YYYY SALES_AMOUNT

         0       2001   5.9598E+11
         0       2002   3123737106
         1       2001   6.3789E+11
         2       2001   6.3903E+11
         2       2002   3538489159
         3       2001   6.4069E+11
         4       2001   6.3885E+11
         4       2002   3537548948
         5       2001   6.0365E+11

9 rows selected.

Execution Plan

   0      SELECT STATEMENT Optimizer=CHOOSE  (Cost=11 Card=9 Bytes=576)
   1    0   SORT  (GROUP BY) (Cost=11 Card=9 Bytes=576)
   2    1     HASH JOIN  (Cost=9 Card=78 Bytes=4992)
   3    2       HASH JOIN  (Cost=6 Card=78 Bytes=4446)
   4    3         VIEW  (Cost=3 Card=19 Bytes=133)
   5    4           SORT  (UNIQUE)  (Cost=3 Card=19 Bytes=133)
   6    5             INDEX  (FAST FULL SCAN) OF 'SYS_IOT_TOP_30180' (U
   7    3         TABLE ACCESS  (FULL) OF 'SALES_MV (Cost=1 Card=409 B
   8    2       VIEW  (Cost=3 Card=100 Bytes=700)
   9    8         SORT  (UNIQUE)  (Cost=3 Card=100 Bytes=700)
  10    9           INDEX  (FULL SCAN) OF 'SYS_IOT_TOP_30185'  (UNIQUE)

Statistics

          0  recursive calls
         16  db block gets
         14  consistent gets
```

Оказывается, что сервер Oracle смог использовать **обе** иерархии измерения и выполнил запрос к материализованному представлению. Благодаря созданным измерениям он выполнил простой поиск для преобразования значения столбца **CUST_ID** в **REGION** (поскольку значение **CUST_ID** определяет значение **ZIP_CODE,** а оно, в свою очередь,

определяет **REGION),** значения столбца **MMYYYY —** в **QTR_YYYY** и ответил на запрос почти моментально. Здесь нам удалось сократить количество операций логического ввода-вывода с более чем 700000 до 16. Если учесть, что таблица **SALES** со временем будет расти, а размер представления **SALES_MV** будет увеличиваться намного медленнее (примерно 180 записей в месяц), запрос будет очень хорошо масштабироваться.

Пакет DBMS_OLAP

Последним фрагментом головоломки, которую представляют собой материализованные представления, является пакет **DBMS_OLAP.** Этот пакет используется для:

- оценки размера материализованного представления в строках и байтах;
- проверки корректности объектов-измерений, с учетом заданных отношений первичного/внешнего ключа;
- получения рекомендаций о создании дополнительных материализованных представлений и поиска лишних, которые надо удалить, с учетом их реального использования и структуры или только структуры;
- оценки использования материализованного представления с помощью предоставляемых процедур, которые информируют о фактической полезности имеющихся материализованных представлений независимо от того, использовались они или нет.

К сожалению, процедуры оценки полезности выходят за рамки тем, которые я могу раскрыть в одной главе. Для использования этих процедур необходимо настроить утилиту Oracle Trace и средства Enterprise Manager Performance Pack, но остальные три процедуры мы рассмотрим.

Чтобы использовать пакет **DBMS_OLAP,** необходимо настроить использование внешних процедур, поскольку большая часть кода пакета **DBMS_OLAP** хранится в библиотеке, написанной на языке C. Если выдается сообщение об ошибке следующего вида, выполните инструкции по настройке, представленные в главе 18:

```
ERROR at line 1:
ORA-28575: unable to open RPC connection to external procedure agent
ORA-06512: at "SYS.DBMS_SUMADV", line 6
ORA-06512: at "SYS.DBMS_SUMMARY", line 559
ORA-06512: at line 1
```

Оценка размера

Процедура **ESTIMATE_SUMMARY_SIZE** информирует о предположительном количестве строк и размере в байтах материализованного представления. Поскольку ретроспективный анализ дает наилучшие результаты, можно оценить эти значения с помощью пакета **DBMS_OLAP**, а затем сравнить с реальными значениями.

Для запуска процедуры необходимо убедиться, что в схеме установлена таблица **PLAN_TABL**E. Соответствующий оператор **CREATE TABLE** можно найти в файле **[ORACLE_HOME]/rdbms/admin/utlxplan.sql** на сервере. При выполнении этого сцена-

рия автоматически будет создана таблица **PLAN_TABLE**. Эта таблица используется при выполнении оператора **EXPLAIN PLAN**, результаты работы которого, в свою очередь, используются пакетом **DBMS_OLAP** для оценки размера материализованного представления. При наличии этой таблицы можно использовать встроенную процедуру **ESTIMATE_SUMMARY_SIZE** для оценки количества строк/байтов в материализованном представлении, которое предполагается создать. Я начну с удаления статистической информации (**DELETE STATISTICS**) о материализованном представлении **SALES_MV**. Обычно пакету **DBMS_OLAP** недоступно материализованное представление, размер которого оценивается, поэтому нам придется этот размер скрыть (в противном случае пакет **DBMS_OLAP** получит точный ответ по словарю данных):

```
tkyte@TKYTE816> analyze table sales_mv DELETE statistics;
Table analyzed.

tkyte@TKYTE816> declare
  2     num_rows number;
  3     num_bytes number;
  4  begin
  5     dbms_olap.estimate_summary_size
  6       ('SALES_MV_ESTIMATE',
  7         'select customer_hierarchy.zip_code,
  8                 time_hierarchy.mmyyyy,
  9                sum(sales.sales_amount) sales_amount
 10           from sales, time_hierarchy, customer_hierarchy
 11          where sales.trans_date = time_hierarchy.day
 12            and sales.cust_id = customer_hierarchy.cust_id
 13          group by customer_hierarchy.zip_code, time_hierarchy.mmyyyy',
 14         num_rows,
 15         num_bytes);
 16
 17      dbms_output.put_line(num_rows || ' rows');
 18     dbms_output.put_line(num_bytes || ' bytes');
 19  end;
 20  /
409 rows
36401 bytes

PL/SQL procedure successfully completed.
```

Первый параметр процедуры — имя плана, под которым его надо запомнить в таблице планов. Это имя не имеет особого значения, но его надо задать в операторе DELETE **FROM PLAN_TABLE WHERE STATEMENT_ID = 'SALES_MV_ESTIMATE'** по завершении эксперимента. Второй параметр — запрос, который будет использоваться для создания материализованного представления. Пакет **DBMS_OLAP** проанализирует этот запрос на основе статистической информации о базовых таблицах, чтобы оценить размер этого объекта. Остальные два параметра предназначены для передачи результатов работы процедуры пакета **DBMS_OLAP** — предполагаемого количества строк и байтов. Они получили значения 409 и 36401, соответственно. Теперь давайте подсчитаем реальные значения:

```
tkyte@TKYTE816> analyze table sales_mv COMPUTE statistics;
Table analyzed.

tkyte@TKYTE816> select count(*) from sales_mv;

  COUNT(*)

      1250

tkyte@TKYTE816> select blocks * 8 * 1024
  2    from user_tables
  3   where table_name = 'SALES_MV'
  4   /

BLOCKS*8*1024

        40960
```

Итак, процедура **ESTIMATE_SUMMARY_SIZE** дала хороший результат при оценке размера таблицы, но недооценила количество строк. Обычно с оценками так и происходит: какие-то параметры оцениваются верно, а какие-то — нет. Я бы использовал эту процедуру для грубой оценки предполагаемого размера объекта.

Проверка достоверности измерений

Эта процедура проверяет достоверность иерархий, входящих в указанное измерение. Так, в рассмотренном ранее примере она проверит, действительно ли значение **CUST_ID** определяет значение **ZIP_CODE**, а то, в свою очередь, определяет значение столбца **REGION**. Чтобы увидеть соответствующую процедуру в действии, создадим пример недостоверного измерения. Начнем с таблицы, содержащей строку для каждого дня года с атрибутами день, месяц и год:

```
tkyte@TKYTE816> create table time_rollup
  2   (day      date,
  3    mon      number,
  4    year     number
  5   )
  6   /

Table created.

tkyte@TKYTE816> insert into time_rollup
  2 select dt, to_char(dt,'mm'), to_char(dt,'yyyy')
  3   from (select trunc(sysdate,'year')+rovmum-1 dt
  4           from all_objects where rownum < 366)
  5   /

365 rows created.
```

Итак, мы создали развернутую информацию по дате, аналогично предыдущему примеру. На этот раз, однако, я не включил год в атрибут, представляющий месяц, — только две цифры, представляющие порядковый номер месяца в году. Если добавить в эту таблицу еще одну строку:

```
tkyte@TKYTE816> insert into time_rollup values
  2  (add_months(sysdate,12),
  3   to_char(add_months(sysdate,12),'mm'),
  4   to_char(add_months(sysdate, 12), 'yyyy'));

1 row created.
```

проблема станет очевидной. Мы будем утверждать, что значение **DAY** определяет значение **MON,** а **MON,** в свою очередь, определяет значение **YEAR, но** в данном случае это неверно. Одно и то же значение месяца будет в таблице для **двух** разных годов. Пакет **DBMS_OLAP** позволяет проверить достоверность; при этом ошибка будет выявлена. Сначала создаем измерение:

```
tkyte@TKYTE816> create dimension time_rollup_dim
  2     level day is time_rollup.day
  3     level mon is time_rollup.mon
  4     level year is time_rollup.year
  5  hierarchy time_rollup
  6  (
  7          day child of mon child of year
  8  )
  9  /

Dimension created.
```

А затем проверяем его достоверность:

```
tkyte@TKYTE816> exec dbms_olap.validate_dimension('time_rollup_dim',
user, false, false);

PL/SQL procedure successfully completed.
```

Кажется, что все в порядке, но надо проверить таблицу, автоматически **созданную** и заполненную в процессе работы с данными:

```
tkyte@TKYTE816> select * from mviewS_exceptions;

OWNER TABLE_NAME   DIMENSION_NAME   RELATIONSHI BAD_ROWID

TKYTE TIME_ROLLUP  TIME_ROLLUP_DIM  CHILD OF    AAAGkxAAGAAAAcKAA7
TKYTE TIME_ROLLUP  TIME_ROLLUP_DIM  CHILD OF    AAAGkxAAGAAAAcKAA8
TKYTE TIME_ROLLUP  TIME_ROLLUP_DIM  CHILD OF    AAAGkxAAGAAAAcKAA9


32 rows selected.
```

Если просмотреть строки, на которые указывает представление **MVIEW$_EXCEPTIONS**, окажется, что это строки за март (я выполнял этот пример в марте). А именно:

```
tkyte@TKYTE816> select * from time_rollup
  2  where rowid in (select bad_rowid from mview$_exceptions);

DAY               MON        YEAR

01-MAR-01           3        2001
02-MAR-01           3        2001
```

```
03-MAR-01          3       2001
04-MAR-01          3       2001

30-MAR-01          3       2001
31-MAR-01          3       2001
26-MAR-02          3       2002

32 rows selected.
```

Теперь проблема ясна: значение **MON** не определяет однозначно **YEAR** — измерение недостоверно. Его использование небезопасно, поскольку будет получен неверный ответ.

Рекомендуется проверять достоверность измерений после их изменения, чтобы гарантировать целостность результатов, получаемых из материализованных представлений благодаря наличию этих измерений.

Рекомендация создания материализованных представлений

Один из наиболее интересных вариантов использования пакета **DBMS_OLAP** — определение того, какие материализованные представления имеет смысл создавать. Процедура **RECOMMEND** делает именно это.

Имеется две версии этой процедуры.

- Процедура **RECOMMEND_MV** анализирует структуру таблицы, внешние ключи, материализованные представления, всю соответствующую статистическую информацию, а затем выдает список рекомендаций в порядке убывания приоритетов.
- Процедура **RECOMMEND_MV_W** идет еще дальше. Если используется утилита Oracle Trace и Enterprise Manager Performance Packs, процедура анализирует запросы, обрабатываемые системой, и рекомендует создание материализованных представлений на основе информации о реальной работе.

В качестве простого примера, оценим с помощью пакета **DBMS_OLAP** существующую "таблицу фактов" **SALES**.

*Таблица фактов (таблица основной информации) — это таблица в схеме "звезда", содержащая фактические данные. Использованная ранее таблица **SALES** — это таблица фактов. В таблице фактов обычно есть два типа столбцов: столбцы, содержащие факты (например, столбец **SALES_AMOUNT** в нашей таблице **SALES**), и столбцы, являющиеся внешними ключами к таблицам измерений (например, к таблице **TRANS_DATE** для нашей таблицы **SALES**).*

Давайте посмотрим, что порекомендует пакет **DBMS_OLAP**. Сначала необходимо создать внешние ключи. Процедуры **RECOMMEND** не будут анализировать объекты DIMENSION при выработке рекомендаций — для определения связей между таблицами им необходимы внешние ключи:

```
tkyte@TKYTE816> alter table sales add constraint t_fk_time
  2  foreign key( trans_date) references time_hierarchy
  3  /

Table altered.

tkyte@TKYTE816> alter table sales add constraint t_fk_cust
  2  foreign key( cust_id) references customer_hierarchy
  3  /

Table altered.
```

После этого можно анализировать нашу таблицу фактов, SALES:

```
tkyte@TKYTE816> exec dbms_olap.recommend_mv('SALES', 10000000000, '' );

PL/SQL procedure successfully completed.
```

Мы попросили процедуру **RECOMMEND_MV**:

- проанализировать таблицу **SALES**;
- учесть, что для материализованных представлений места предостаточно (мы просто передали очень большое значение в качестве ограничения);
- не оставлять без необходимости существующих материализованных представлений (мы передали " в качестве списка имен сохраняемых представлений).

Теперь можно либо непосредственно обращаться к заполненным этой процедурой таблицам, либо, что удобнее, использовать простую процедуру для выдачи их содержимого. Для установки этой процедуры и построения отчета надо выполнить:

```
tkyte@TKYTE816> @C:\oracle\RDBMS\demo\saclvdemo

Package created.

Package body created.

Package created.

Package body created.

tkyte@TKYTE816> exec demo_sumadv.prettyprint_recornmendations
Recommendation Number = 1
Recommended Action is CREATE new summary:
SELECT CUSTOMER_HIERARCHY.CUST_ID, CUSTOMER_HIERARCHY.ZIP_CODE,
CUSTOMER_HIERARCHY.REGION , COUNT(*), SUM(SALES.SALES_AMOUNT),
COUNT(SALES.SALES_AMOUNT)
FROM TKYTE.SALES, TKYTE.CUSTOMER_HIERARCHY
WHERE SALES.CUST_ID = CUSTOMER_HIERARCHY.CUST_ID
GROUP BY CUSTOMER_HIERARCHY.CUST_ID, CUSTOMER_HIERARCHY.ZIP_CODE,
CUSTOMER_HIERARCHY.REGION
Storage in bytes is 2100
Percent performance gain is 43.2371266138567
Benefit-to-cost ratio is .0205891079113603
Recommendation Number = 2


PL/SQL procedure successfully completed.
```

Процедура пакета **DBMS_OLAP** учла существующие измерения **и** материализованные представления и выдала рекомендации по созданию дополнительных материализованных представлений, которые могут оказаться полезными с учетом имеющихся у сервера метаданных (первичных ключей, внешних ключей и измерений).

При использовании утилиты Oracle Trace можно пойти в процессе выработки рекомендаций чуть дальше. Утилита Oracle Trace позволяет перехватывать **фактические** запросы к данным, принимаемые сервером, и записывать подробную информацию о них. Эта информация будет использоваться пакетом **DBMS_OLAP** для выработки еще более точных рекомендаций, основанных не только на потенциальных возможностях, но и на реальных запросах, выполняемых к данным. При этом не рекомендуется создание материализованных представлений, которые теоретически могли бы, но не будут использоваться при таком потоке запросов. Рекомендоваться будут только те материализованные представления, которые будут использованы для выполнения запросов, реально поступающих в систему.

Проблемы

При использовании материализованных представлений надо учитывать ряд соображений. Мы кратко поговорим о них в этом разделе.

Материализованные представления не предназначены для систем OOT

Как уже упоминалось, поддержка материализованных представлений требует дополнительных затрат ресурсов при выполнении отдельных транзакций и, если представления созданы с опцией **REFRESH ON COMMIT,** приводит к конфликтам. Дополнительные затраты ресурсов связаны с необходимостью учета изменений, выполненных транзакцией, — эти изменения будут либо сохраняться в качестве данных состояния сеанса, либо регистрироваться в таблицах. В системах, интенсивно обрабатывающих транзакции, такие дополнительные затраты ресурсов нежелательны. Проблема одновременного доступа возникает для материализованных представлений с опцией **REFRESH ON COMMIT** из-за того, что многие записи в исходной таблице фактов ссылаются на одну строку в итоговой таблице. Изменение любой из многих тысяч записей, которые могут существовать, приводит к необходимости изменить одну строку в итоговой таблице. Это, естественно, мешает одновременному доступу при интенсивных изменениях.

Это не исключает использования материализованных представлений, в частности представлений, обновляемых по требованию (**REFRESH ON DEMAND**), в среде OOT при *полном обновлении*. Полное обновление не требует дополнительных затрат ресурсов на отслеживание изменений на уровне транзакций. Вместо этого в определенный момент времени выполняется запрос, определяющий материализованное представление, и его результаты просто заменяют существующее материализованное представление. Поскольку делается это по требованию (или периодически), такое обновление можно выполнять в периоды низкой загруженности сервера. Полученное материализованное

представление особенно пригодится для создания отчетов: данные OOT можно каждую ночь преобразовывать с помощью SQL-операторов в форму, упрощающую и ускоряющую выполнение запросов. На следующий день оперативные отчеты по результатам вчерашней работы выполняются максимально быстро, не мешая при этом обработке транзакций.

Целостность запросов при переписывании

Как было описано ранее, есть три режима обеспечения целостности.

- **ENFORCED.** Будет использовать материализованное представление, только если при этом невозможно получение некорректных или устаревших данных.
- **TRUSTED.** Сервер Oracle будет использовать материализованное представление, даже если некоторые требования, на которые он при этом полагается, им не проверяются и не обеспечиваются. Эта ситуация типична в среде хранилища данных, где многие требования целостности соблюдены, но не поддерживаются сервером Oracle.
- **STALE_TOLERATED.** Сервер Oracle будет использовать материализованное представление, даже если "знает", что данные, по которым оно построено, изменились. Эта ситуация типична для среды создания отчетов, вроде описанной выше.

Надо понимать последствия использования каждого из этих режимов. Режим **ENFORCED** дает правильные ответы всегда — за счет отказа от использования некоторых материализованных представлений, которые могли бы ускорить выполнение запроса. В режиме **TRUSTED**, если окажется, что условие, которому сервер Oracle попросили доверять, в действительности не выполняется, могут выдаваться результаты, отличные от получаемых по исходным данным. Такая ситуация была рассмотрена в примере с материализованным представлением **EMP_DEPT.** Режим **STALE_TOLERATED** следует использовать в системах создания отчетов, где вполне допустимо получить устаревшее значение. Если требуется актуальная информация с точностью до минуты, режим **STALE_TOLERATED** использовать нельзя.

Резюме

Материализованные представления — мощное средство повышения производительности хранилищ данных и систем поддержки принятия решений. Одно материализованное представление может использоваться многими взаимосвязанными запросами. Самое главное, что его использование полностью прозрачно для приложения и пользователя. Не нужно сообщать пользователям, какие итоговые таблицы поддерживаются, - об этом информируется сервер Oracle с помощью требований целостности ссылок и измерений. Все остальное сервер сделает автоматически.

Материализованные представления — естественное развитие и объединение различных возможностей сервера и средств поддержки принятия решений. Средства поддержки итоговых таблиц Oracle Discoverer (и других подобных программ) уже не ограничены только этой средой. Теперь любой клиент, от замечательного SQL*Plus до разработан-

ного вами приложения и готовых средств создания отчетов, может использовать уже где-то хранящийся ответ на его запрос.

Добавьте к этому возможности пакета **DBMS_OLAP**. Он не только позволяет оценить, сколько дополнительно места на диске понадобится для поддержки материализованного представления, но и может следить за использованием существующих представлений. На основе этой информации пакет рекомендует удалить одни и создать другие представления — вплоть до выдачи текста запроса, который имеет смысл для этого использовать.

В конечном итоге, материализованные представления в среде, где данные только читаются или интенсивно читаются, несомненно, оправдают выделенное для них дополнительное место на диске, сократив время выполнения и экономя ресурсы, необходимые для фактического выполнения запросов.

14

Фрагментация

Возможность *фрагментации,* то есть разбиения таблицы или индекса на несколько меньших, проще управляемых частей, впервые появилась в сервере Oracle версии 8.0. Логически для обращающегося к базе данных приложения есть только одна таблица или индекс. Физически же эта таблица или индекс могут состоять из многих десятков фрагментов. Каждый фрагмент — самостоятельный объект, с которым можно работать отдельно или как с частью большего объекта.

Фрагментация разрабатывалась для упрощения управления очень большими таблицами и индексами за счет применения подхода "разделяй и властвуй". Предположим, в базе данных имеется индекс размером 10 Гбайт. Если необходимо перестроить этот нефрагментированный индекс, придется перестраивать весь индекс в один прием. Хотя сервер способен перестраивать такие индексы динамически, объем ресурсов, необходимых для полного пересоздания всего индекса размером 10 Гбайт, огромен. Потребуется еще не менее 10 Гбайт свободного пространства для хранения обоих экземпляров индекса, необходима временная таблица журнала транзакций для записи изменений, сделанных в базовой таблице за время пересоздания индекса, и т.д. С другой стороны, если индекс разбит на десять фрагментов размером 1 Гбайт, можно пересоздавать каждый фрагмент индекса отдельно, по одному. При этом потребуется только 10 процентов свободного пространства, которое понадобилось бы при пересоздании нефрагментированного индекса. Пересоздание индекса пройдет намного быстрее (возможно, раз в десять), намного уменьшится объем выполненных транзакциями изменений, которые придется учесть в новом индексе, и т.д.

Короче, фрагментация может сделать устрашающие по поглощению ресурсов, а иногда даже невозможные в большой базе данных операции настолько же простыми, как в маленькой.

Использование фрагментации

Для использования фрагментации имеются три причины:

- повышение доступности данных;
- упрощение администрирования;
- повышение производительности запросов и операторов ЯМД.

Повышение доступности данных

Повышение доступности данных достигается за счет того, что фрагменты являются независимыми сущностями. Доступность (или недоступность) одного фрагмента объекта не означает, что весь объект доступен (или недоступен). Оптимизатор учтет реализованную схему фрагментации и удалит из плана выполнения запроса фрагменты, данные из которых не запрашиваются. Если недоступен один фрагмент большого объекта и запрос не обращается к данным в этом фрагменте, сервер Oracle успешно выполнит этот запрос. Давайте создадим фрагментированную по хеш-функции таблицу из двух фрагментов, хранящихся в разных табличных пространствах, и вставим в нее данные. Для каждой вставляемой в таблицу строки значение в столбце **EMPNO** хешируется с целью определения фрагмента (а значит, и табличного пространства, в данном случае), в который следует помещать данные. Затем, дополнив имя таблицы именем фрагмента, просмотрим содержимое каждого фрагмента:

```
tkyte@TKYTE816> CREATE TABLE emp
  2  (empno   int,
  3   ename   varchar2(20)
  4  )
  5  PARTITION BY HASH (empno)
  6  (partition part_1 tablespace p1,
  7   partition part_2 tablespace p2
  8  )
  9  /

Table created.

tkyte@TKYTE816> insert into emp select empno, ename from scott.emp
  2  /

14 rows created.

tkyteeTKYTE816> select * from emp partition(part_1);

     EMPNO ENAME

      7369 SMITH
      7499 ALLEN
      7654 MARTIN
      7698 BLAKE
```

```
      7782 CLARK
      7839 KING
      7876 ADAMS
      7934 MILLER
8 rows selected.

tkyte@TKYTE816> select * from emp partition(part_2);

     EMPNO ENAME

      7521 WARD
      7566 JONES
      7788 SCOTT
      7844 TURNER
      7900 JAMES
      7902 FORD
6 rows selected.
```

Теперь сделаем часть данных недоступными, отключив одно из используемых табличных пространств, и выполним запрос, затрагивающий оба фрагмента, чтобы показать невозможность его выполнения. Вы увидите, что запрос, не обращающийся к отключенному табличному пространству, работает как обычно — сервер Oracle не обращается к отключенному табличному пространству. В этом примере я использовал связываемую переменную, чтобы продемонстрировать, что, даже если в момент оптимизации запроса сервер не знает, к какому фрагменту будут обращаться, он все равно пропустит ненужный фрагмент:

```
tkyteeTKYTE816> alter tablespace p1 offline;

Tablespace altered.

tkyte@TKYTE816> select * from emp
  2  /
select * from emp

ERROR at line 1:
ORA-00376: file 4 cannot be read at this time
ORA-01110: data file 4: 'C:\ORACLE\ORADATA\TKYTE816\P1.DBF'

tkyteeTKYTE816> variable n number
tkyte@TKYTE816> exec :n := 7844

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select * from emp where empno = :n
  2  /

     EMPNO ENAME

      7844 TURNER
```

Как видите, мы отключили одно из табличных пространств, сымитировав сбой диска. В результате, если потребуется обратиться ко всей таблице, естественно, ничего не получится. Однако обратиться к данным, находящимся в доступном фрагменте, можно. Когда оптимизатор может удалить фрагменты из плана выполнения запроса, он это де-

лает. Этот факт увеличивает доступность данных для приложений, использующих в запросах ключ, по которому выполнена фрагментация.

Фрагменты повышают доступность также благодаря сокращению времени простоя. Например, если таблица размером 100 Гбайт разбита на 50 фрагментов размером 2 Гбайт, восстановление в случае ошибок выполняется в 50 раз быстрее. Если один из фрагментов размером 2 Гбайта поврежден, для его восстановления нужно намного меньше времени, чем для восстановления таблицы размером 100 Гбайт. Таким образом, доступность повышается по двум направлениям: многие пользователи могут вообще не заметить, что данные были недоступны, благодаря тому, что сбойный фрагмент пропускается, а время простоя при сбое сокращается вследствие существенно меньшего объема работы, необходимой для восстановления.

Упрощение администрирования

Упрощение администрирования связано с тем, что операции с маленькими объектами выполнять гораздо проще, быстрее и при этом требуется меньше ресурсов, чем в случае больших объектов. Например, если оказалось, что 50 процентов строк в таблице фрагментированы (подробнее о фрагментации и переносе строк см. в главе 6, посвященной таблицам), и необходимо это исправить, фрагментация таблицы очень пригодится. Чтобы исключить фрагментацию строк, как правило, пересоздается объект, в данном случае — таблица. Если таблица размером 100 Гбайт, придется выполнять эту операцию одним большим "куском", последовательно, с помощью оператора **ALTER TABLE MOVE.** Если же эта таблица разбита на 25 фрагментов размером 4 Гбайта, можно перестраивать фрагменты по одному. Более того, если это делается в период минимальной загруженности сервера, можно даже выполнять операторы **ALTER TABLE MOVE** параллельно в отдельных сеансах, что позволяет сократить время пересоздания. Практически все, что делается с нефрагментированным объектом, можно сделать и с частью фрагментированного объекта.

Еще один фактор, который необходимо учитывать при оценке влияния фрагментации на администрирование, — использование смещающегося окна данных в хранилищах данных и при архивировании. Во многих случаях необходимо сохранять доступными последние (по времени создания) N групп данных. Например, необходимо предоставлять данные за последние 12 месяцев или пять лет. При отсутствии фрагментации это обычно связано с множественной вставкой новых данных, а затем — множественным удалением устаревших. Для этого необходимо выполнить множество операторов ЯМД, сгенерировать множество данных повторного выполнения и отката. При использовании фрагментации можно:

- загрузить в отдельную таблицу данные за новый месяц (или год, или любой другой период);
- полностью проиндексировать таблицу (эти шаги можно сделать вообще в другом экземпляре, а результаты перенести в текущую базу данных);
- добавить ее **в конец** фрагментированной таблицы;
- удалить самый старый фрагмент с другого конца фрагментированной таблицы.

Итак, теперь легко поддерживать очень большие объекты, содержащие хронологическую информацию. Устаревшие данные просто **удаляются** из фрагментированной таблицы, если они не нужны, или помещаются в архив. Новые данные можно загрузить в отдельную таблицу, чтобы не мешать доступу к фрагментированной таблице во время загрузки, индексирования и т.п. Полный пример использования смещающегося окна будет представлен далее.

Повышение производительности операторов ЯМД и запросов

Еще одним преимуществом фрагментации является повышение производительности запросов и операторов ЯМД. Мы рассмотрим преимущества фрагментации для этих двух категорий операторов отдельно.

Повышение производительности операторов ЯМД связано с потенциальной возможностью распараллеливания. При распараллеливании операторов ЯМД сервер Oracle использует несколько потоков или процессов для выполнения операторов **INSERT**, **UPDATE** или **DELETE**. На многопроцессорной машине с большой пропускной способностью ввода-вывода потенциальное ускорение для операторов ЯМД, выполняющих множественные изменения, может быть весьма большим. В отличие от параллельных запросов (обработки несколькими процессами/потоками оператора **SELECT**), для распараллеливания операторов ЯМД требуется фрагментация (есть специальный случай параллельной непосредственной вставки, задаваемой с помощью подсказки /*+ **APPEND** */> когда фрагментация не требуется). Если таблицы не фрагментированы, распараллелить операторы ЯМД не удастся. Сервер Oracle присваивает каждому объекту максимальную степень распараллеливания в зависимости от количества составляющих его фрагментов.

Не стоит ожидать от распараллеливания операторов ЯМД ускорения работы приложений оперативной обработки транзакций (OOT). В этом отношении есть много заблуждений. Я много раз слышал: "Параллельно выполняемые операции должны давать результаты быстрее, чем при последовательном выполнении". Это не всегда верно. При параллельном выполнении некоторых операций требуется во много раз больше времени, чем при последовательном. На организацию параллельного выполнения расходуются определенные ресурсы, необходима дополнительная координация. Более того, распараллеливание вообще не стоит использовать в среде интенсивной оперативной обработки транзакций — в этом просто нет смысла. Распараллеливание операций предназначено исключительно для максимального использования ресурсов системы. При параллельном выполнении один пользователь может единолично использовать все диски, процессоры и всю свободную память машины. В среде хранилища данных, где данных много, а пользователей мало, именно это и требуется. В системе OOT (большое количество пользователей, выполняющих короткие, быстрые транзакции) предоставление пользователю возможности полностью использовать ресурсы машины отрицательно скажется на масштабируемости.

В этом есть кажущееся противоречие: мы используем распараллеливание запроса для ускорения работы с большими объемами данных, как же это решение может быть немасштабируемым? Применительно к системе OOT, однако, это утверждение абсолютно

верно. Параллельные запросы плохо масштабируются при увеличении количества пользователей, одновременно их выполняющих. Параллельные запросы позволяют в одном пользовательском сеансе выполнять столько же действий, как в ста одновременных сеансах. Однако в системе ООТ нежелательно, чтобы один пользователь выполнял столько же действий, как сто.

Распараллеливание операторов ЯМД полезно в среде больших хранилищ данных как средство множественного изменения больших объемов данных. Распараллеленный оператор ЯМД выполняется сервером во многом аналогично параллельному запросу: каждый фрагмент используется как отдельный экземпляр базы данных. Каждый фрагмент изменяется отдельным потоком в отдельной транзакции (поэтому при изменении возможно использование отдельного сегмента отката), а когда все изменения закончатся, происходит нечто подобное быстрой двухфазной фиксации отдельных, независимых транзакций. В связи с такой архитектурой при распараллеливании операторов ЯМД возникает ряд ограничений. Например, в ходе параллельного выполнения оператора ЯМД не поддерживаются триггеры. Это, по-моему, разумное ограничение, поскольку выполнение триггеров при изменении обычно существенно увеличивает использование ресурсов, а распараллеливание выполняется для максимального ускорения — это несовместимые действия. Кроме того, при распараллеливании операторов ЯМД не поддерживается ряд декларативных требований целостности ссылок, поскольку каждый фрагмент обрабатывается в отдельной транзакции, по сути — в отдельном сеансе. Не поддерживается, например, целостность ссылок объекта на самого себя. Представьте себе проблемы блокирования и взаимные блокировки, которые могли бы возникнуть при выполнении таких требований.

С точки зрения производительности фрагментация обеспечивает две специализированные операции.

- **Игнорирование фрагмента.** Некоторые фрагменты данных не просматриваются при обработке запроса. Пример игнорирования фрагмента был представлен ранее, когда мы отключили одно из табличных пространств и смогли, тем не менее, выполнить запрос. Отключенное табличное пространство при выполнении запроса было пропущено.
- **Параллельное выполнение операций.** Распараллеливать можно соединения по фрагментам, если объекты фрагментированы по ключам соединения, или просмотр индексов.

Как и в случае распараллеливания операторов ЯМД, не стоит ожидать от фрагментации существенного повышения производительности в среде ООТ. Игнорирование фрагмента эффективно при полном просмотре больших объектов. Пропуская фрагмент, можно избежать просмотра больших частей объекта. Именно за счет этого может повышаться производительность. В среде ООТ, однако, большие объекты **полностью** не **просматриваются** (если просматриваются — это серьезная ошибка проектирования). Даже если фрагментировать индексы, ускорение при просмотре меньшей части индекса будет незначительным (или его вообще не будет). Если часть запросов использует индекс **и** при поиске можно пропустить только один фрагмент индекса запросы после такой фрагментации могут выполняться медленнее, поскольку вместо одного большого индекса

теперь надо просматривать 5, 10 или 20 маленьких. Более детально мы изучим это позже, при рассмотрении типов фрагментации индексов. Имеется возможность повысить эффективность работы системы ООТ с помощью фрагментации, например, повышая степень параллелизма за счет уменьшения количества конфликтов. Фрагментацию можно использовать для распределения изменений одной таблицы по нескольким физическим фрагментам. Вместо использования одного сегмента таблицы и одного сегмента индекса можно создать 20 фрагментов таблицы и 20 фрагментов индекса. Результат будет таким же, как при наличии 20 таблиц вместо одной, — конфликтов при изменениях этого разделяемого ресурса станет меньше.

Что касается распараллеливания запросов, то оно, как я уже подчеркивал ранее, в среде ООТ нежелательно. Распараллеливание операций имеет смысл оставить администратору базы данных для пересоздания и создания индексов, анализа таблиц и т.п. Для запросов в системе ООТ обычно характерен очень быстрый доступом к данным по индексу, так что фрагментация не слишком ускорит этот доступ. Это не означает, что следует вообще отказаться от использования фрагментации в среде ООТ, просто не стоит ожидать существенного повышения производительности только за счет фрагментации объектов. Приложения в данном случае не могут использовать преимущества, которые могла бы дать фрагментация.

В среде же хранилища данных/систем поддержки принятия решений (СППР) фрагментация не только существенно упрощает администрирование, но и позволяет ускорить работу. Предположим, имеется большая таблица, к которой надо выполнить запрос, анализирующий продажи по кварталам. Для каждого квартала имеются сотни тысяч записей, а общее количество записей измеряется миллионами. Итак, необходимо запросить сравнительно небольшой срез общего набора данных, но индексировать данные по кварталам нет смысла. Такой индекс будет ссылаться на сотни тысяч записей для каждого квартала, и просмотр индекса по диапазону при этом будет выполняться чрезвычайно медленно (подробнее об этом см. в главе 7, посвященной индексам). Итак, для выполнения большого количества запросов приходится полностью просматривать таблицу, но при этом просматриваются миллионы записей, большинство из которых не связаны с выполняемым запросом. Используя соответствующую схему фрагментации, можно разделить данные по кварталам таким образом, что при запросе данных за конкретный квартал полностью просматривать придется только один фрагмент. Это лучшее из всех возможных решений.

Кроме того, в среде хранилищ данных/систем поддержки принятия решений часто используется распараллеливание запросов. Здесь операции типа параллельного просмотра индексов по диапазону или параллельный, быстрый полный просмотр индекса не только имеют смысл, но и дают преимущества. Мы хотим максимально использовать все имеющиеся ресурсы, и распараллеливание запросов позволяет этого добиться. Так что, в подобной среде фрагментация с большой долей вероятности ускорит выполнение запросов.

Упорядочив преимущества фрагментации по степени важности, получим:

1. Увеличение доступности данных — хорошо для всех типов систем.
2. Упрощение администрирования больших объектов базы данных за счет замены их несколькими меньшими — хорошо для всех типов систем.

3. Повышение производительности некоторых операторов ЯМД и запросов — достигается, в основном, в среде больших хранилищ данных.
4. Уменьшение количества конфликтов в системах ООТ с большим количеством вставок (например, в таблицу записей аудита) за счет их распределения по нескольким отдельным фрагментам (распределение "горячей точки" по нескольким дискам).

Как выполняется фрагментация

В этом разделе будут рассмотрены схемы фрагментации, предлагаемые сервером Oracle 8i. Имеется три схемы фрагментации для таблиц и две — для индексов. В рамках двух схем фрагментации индексов можно выделить несколько классов фрагментированных индексов. Мы рассмотрим преимущества и отличительные особенности каждого класса, а также разберемся, какие схемы фрагментации следует применять для различных типов приложений.

Схемы фрагментации таблиц

В настоящее время сервер Oracle поддерживает три способа фрагментации таблиц.

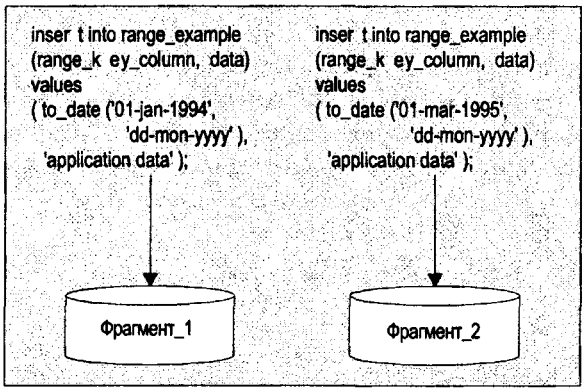
- **Фрагментация по диапазону.** Можно указать диапазоны значений данных, строки для которых должны храниться вместе. Например, все данные за январь 2001 года будут храниться в фрагменте 1, все данные за февраль 2001 года — в фрагменте 2 и т.д. Это, вероятно, самый популярный способ фрагментации в Oracle 8i.
- **Фрагментация по хеш-функции.** Такая схема уже использовалась в первом примере этой главы. К значению одного или нескольких столбцов применяется хеш-функция, определяющая фрагмент, в который помещается строка.
- **Составная фрагментация.** Это сочетание фрагментации по диапазону и по хеш-функции. Можно сначала применить разбиение по диапазону значений данных, а затем выбрать в пределах диапазона фрагмент на основе значения хеш-функции.

Следующий код и схемы наглядно демонстрируют применение этих способов фрагментации. Кроме того, операторы **CREATE TABLE** структурированы так, чтобы можно было понять синтаксис создания фрагментированной таблицы. Сначала рассмотрим таблицу, фрагментированную по диапазону:

```
tkyte@TKYTE816> CREATE TABLE range_example
  2   (range_key_column date,
  3    data             varchar2(20)
  4   )
  5  PARTITION BY RANGE (range_key_column)
  6  (PARTITION part_1 VALUES LESS THAN
  7        (to_date('01-jan-1995','dd-mon-yyyy')),
  8   PARTITION part_2 VALUES LESS THAN
  9   (to_date('01-jan-1996',  'dd-mon-yyyy'))
```

```
10 )
11 /
Table created.
```

Следующая схема показывает, что сервер Oracle проверяет столбец **RANGE_KEY_COLUMN** и в зависимости от его значения вставляет строку в один из фрагментов:



Интересно, что произойдет, если изменится значение столбца, определяющего, в какой фрагмент попадает строка. При этом надо учитывать два случая.

- Изменение не перемещает строку в другой фрагмент; строка принадлежит тому же фрагменту. Такое изменение возможно всегда.
- Изменение вызывает перемещение строки в другой фрагмент. Такое изменение поддерживается, только **если** для таблицы включен перенос строк, иначе выдается сообщение об ошибке.

Эту особенность легко продемонстрировать. Вставим строку во фрагмент **PART_1** созданной ранее таблицы. Затем изменим значение столбца **RANGE_KEY_COLUMN** так, что строка останется во фрагменте **PART_1,** — это изменение будет успешно выполнено. Далее изменим значение столбца **RANGE_KEY_COLUMN** так, чтобы переместить строку во фрагмент **PART_2.** При этом будет выдано сообщение об ошибке, поскольку перенос строк явно включен не был. Наконец, изменим таблицу, разрешив перемещение строк, и продемонстрируем последствия этого изменения:

```
tkyte@TKYTE816> insert into range_exanple
  2  values (to_date('01-jan-1994', 'dd-mon-yyyy'), 'application data'),-
1 row created.
tkyte@TKYTE816> update range_example
  2  set range_key_column = range_key_column+1
  3  /
1 row updated.
```

Как и ожидалось, изменение успешно выполнено, ведь строка осталась во фрагменте **PART_1**. Затем посмотрим, что произойдет, если изменение вызывает перемещение строки:

```
tkyte@TKYTE816>update range_example
  2  set range_key_column = range_key_column+366
  3  /
update range_example
       *
ERROR at line 1:
ORA-14402: updating partition key column would cause a partition change
```

Сразу же выдается сообщение об ошибке. В Oracle 8.1.5 и более новых версиях можно включить поддержку переноса строк для таблицы, что позволит перемещать строку из одного фрагмента в другой. В версиях Oracle 8.0 это было невозможно; приходилось удалять строку и повторно вставлять ее с измененными значениями. Следует, однако, помнить о побочном эффекте перемещения строк. Это — один из двух случаев, когда идентификатор строки (**ROWID**) изменяется при изменении данных (другой — изменение первичного ключа таблицы, организованной по индексу; универсальный идентификатор для этой строки тоже изменится):

```
tkyte@TKYTE816> select rowid from range_example
  2  /

ROWID

AAAHeRAAGAAAAAKAAA

tkyte@TKYTE816> alter table range_example enable row movement
  2  /

Table altered.

tkyte@TKYTE816>update range_example
  2  set range_key_column = range_key_column+366
  3  /

1 row updated.
tkyte@TKYTE816> select rowid from range_example
  2  /

ROWID

AAAHeSAAGAAAABKAAA
```

Итак, если учитывать изменение идентификатора строки при изменении значения ключа фрагментации, включение перемещения строк позволит изменять эти ключи.

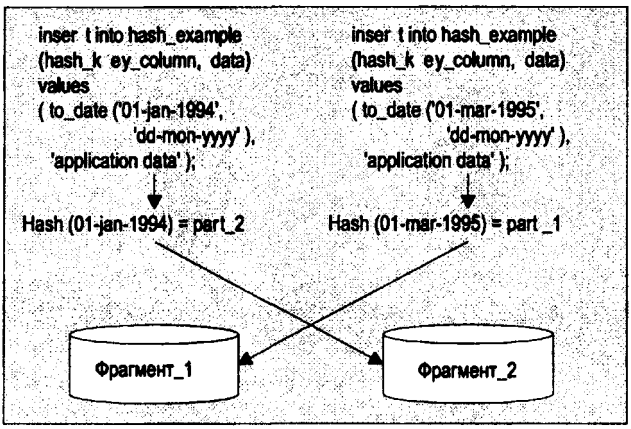
Следующий пример демонстрирует *фрагментацию таблицы по хеш-функции*. В этом случае сервер Oracle будет применять хеш-функцию к ключу фрагментации для определения того, в какой из N фрагментов надо поместить данные. Для наиболее равномерного распределения рекомендуется в качестве значения N использовать степени двойки (2, 4, 8, 16 и т.д.). Фрагментация по хеш-функции предназначена для равномерного распределения данных по нескольким устройствам (дискам). В качестве хеш-ключа для таблицы необходимо выбирать столбец или набор столбцов с как можно большим количеством уникальных значений — это обеспечивает равномерное распределение значений. Если выбрать столбец, имеющий всего четыре значения, и использовать два

фрагмента, в результате хеширования все строки могут оказаться в одном фрагменте, что делает фрагментацию бессмысленной.

Создадим таблицу, распределенную по хеш-функции на два фрагмента:

```
tkyte@TKYTE816> CREATE TABLE hash_example
  2  (hash_key_column   date,
  3   data              varchar2(20)
  4  )
  5  PARTITION BY HASH (hash_key_column)
  6  (partition part_1 tablespace p1,
  7   partition part_2 tablespace p2
  8  )
  9  /

Table created.
```

Следующая схема показывает, что сервер Oracle применит хеш-функцию к столбцу **HASH_KEY_COLUMN** и, в зависимости от ее значения, вставит строку в один из двух фрагментов:

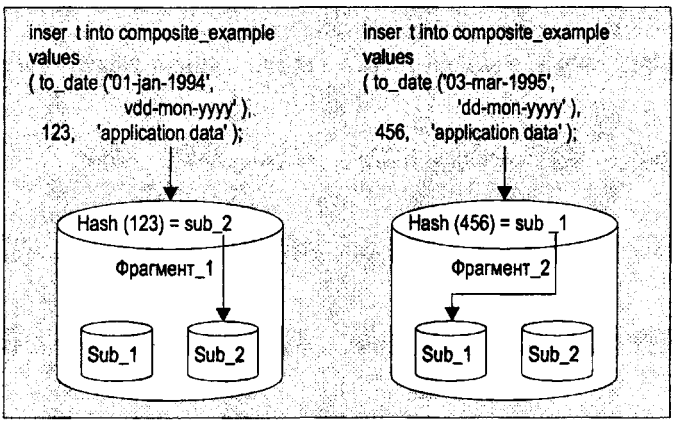


Теперь рассмотрим пример смешанной фрагментации, когда строки фрагментируются и по диапазону, и по хеш-функции. Здесь фрагментация по диапазону будет выполняться для одного набора столбцов, а фрагментация по хеш-функции — для другого. Вполне допустимо использовать одни и те же столбцы в обоих условиях фрагментации:

```
tkyte@TKYTE816> CREATE TABLE composite_example
  2  (range_key_column   date,
  3   hash_key_column    int,
  4   data               varchar2(20)
  5  )
  6  PARTITION BY RANGE (range_key_column)
  7  subpartition by hash(hash_key_column) subpartitions 2
  8  (
  9  PARTITION part_1
 10      VALUES LESS THAN(to_date('01-jan-1995','dd-mon-yyyy'))
 11     (subpartition part_1_sub_1,
 12      subpartition part_1_sub_2
```

```
13      ),
14  PARTITION part_2
15    VALUES LESS THAN(to_date('01-jan-1996','dd-mon-yyyy'))
16      (subpartition part_2_sub_1,
17      subpartition part_2_sub_2
18      )
19  )
20  /

Table created.
```

При смешанной фрагментации сервер Oracle сначала применяет правила фрагментации по диапазону, чтобы понять, к какому диапазону относятся данные, а затем - хеш-функцию, которая и определяет, в какой физический фрагмент попадет строка:



Sub 1 – Подфрагмент 1, Sub_2 – Подфрагмент_2

Фрагментация по диапазону используется, когда данные логически разделяются по значениям. Классический пример — данные, привязанные к периоду времени. Фрагментация по кварталам, по финансовым годам, по месяцам. Фрагментация по диапазону во многих случаях позволяет пропускать фрагменты, в том числе для условий строгого равенства и условий, задающих диапазоны: меньше, больше, в указанных пределах и т.д.

Фрагментация по хеш-функции подходит для данных, в которых не удается выделить естественные диапазоны значений, подходящие для фрагментации. Предположим, необходимо загрузить в таблицу данные переписи населения — в них может и не быть атрибута, по которому имеет смысл разделять данные на диапазоны. Однако хотелось бы воспользоваться преимуществами, которые предоставляет фрагментация с точки зрения администрирования, производительности и доступности данных. Можно выбрать набор столбцов с уникальными значениями, по которым выполнять хеширование. Это позволит равномерно распределить данные по любому количеству фрагментов. Игнорирование фрагмента для объектов, разделенных по хеш-функции, возможно только для условий строгого равенства или **IN (значение 1, значение2,...)**, но не для условий, задающих диапазоны значений.

Составная фрагментация подходит, когда данные логически поделены на диапазоны, но получающиеся в результате фрагменты — слишком большие, чтобы ими можно

было эффективно управлять. Можно разделить данные по диапазону, а затем разбить каждый фрагмент на несколько подфрагментов по хеш-функции. Это позволит распределить операции ввода-вывода по нескольким дискам для каждого большого фрагмента. Кроме того, теперь можно пропускать фрагменты на трех уровнях. Если запрос выполняется по ключу, использованному при фрагментации по диапазону, сервер может пропустить все фрагменты, не отвечающие критериям запроса. Если в запросе будет указан еще и ключ хеширования, сервер сможет пропустить другие подфрагменты в соответствующем диапазоне. Если в запросе используется только ключ хеш-функции (а ключ, по которому выполнена фрагментация по диапазону, не используется), сервер будет обращаться только к соответствующим подфрагментам в каждом диапазоне.

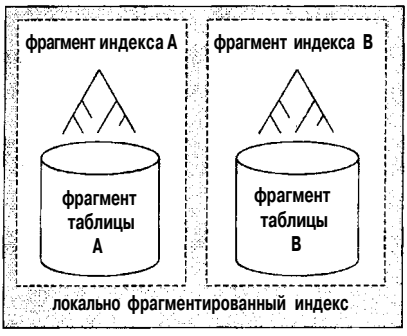
Рекомендуется преимущественно использовать фрагментацию по диапазону, если данные вообще имеет смысл разбивать на диапазоны по каким-то атрибутам. Фрагментация по хеш-функции имеет множество преимуществ, но она не так эффективна с точки зрения возможностей игнорирования фрагментов. Использовать хеш-фрагментацию в пределах фрагментов по диапазону рекомендуется, когда соответствующие диапазонам фрагменты слишком велики для эффективного управления, или в тех случаях, когда желательно распараллеливать операторы ЯМД или иметь возможность параллельного просмотра индексов для отдельного фрагмента.

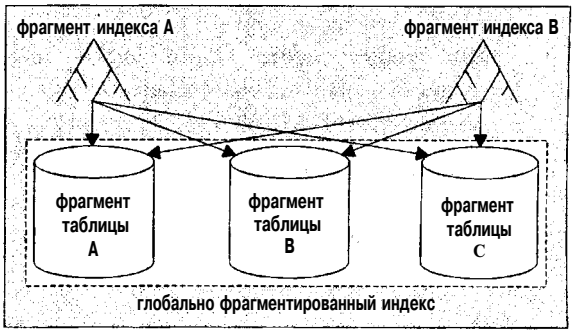
Фрагментация индексов

Индексы, как и таблицы, можно фрагментировать. Существует два способа фрагментации индексов.

- Можно фрагментировать индекс **по тем же критериям, что и базовую таблицу.** Такие индексы называют *локально фрагментированными.* Для каждого фрагмента таблицы будет создан соответствующий фрагмент индекса, индексирующий только этот фрагмент таблицы. Все записи в данном фрагменте индекса ссылаются на один фрагмент таблицы, а все строки фрагмента таблицы представлены в одном фрагменте индекса.
- Можно фрагментировать индекс **по диапазону.** Такие индексы называют *глобально фрагментированными.* Индекс фрагментируется по диапазону, и один фрагмент индекса может ссылаться на любые (хоть **все)** фрагменты базовой таблицы.

Следующие схемы показывают различие между локально и глобально фрагментированными индексами.





Следует помнить, что количество фрагментов глобально фрагментированного индекса может не совпадать с количеством фрагментов таблицы.

Поскольку глобально фрагментированные индексы можно фрагментировать только по диапазону, при разбиении индекса по хеш-функции или в случае составной фрагментации придется использовать локально фрагментированные индексы. Локально фрагментированный индекс использует такую же схему фрагментации, как и базовая таблица.

Локально фрагментированные индексы

Практика показывает, что чаще всего используются локально фрагментированные индексы. Дело в том, что фрагментация, как правило, используется в среде хранилищ данных. В системах OOT более типичны глобально фрагментированные индексы. Локально фрагментированные индексы наиболее подходят для хранилищ данных. Они обеспечивают большую доступность данных (меньшее время простоя), поскольку проблемы доступности, скорее всего, будут связаны с одним диапазоном или хеш-фрагментом данных. Вследствие того что глобально фрагментированный индекс ссылается на несколько фрагментов, для определенных запросов фрагменты могут стать недоступными. Локально фрагментированные индексы обеспечивают большую гибкость при сопровождении фрагментов. Если администратор базы данных решит перенести фрагмент таблицы, изменять придется только соответствующий фрагмент локально фрагментированного индекса. Если индекс фрагментирован глобально, изменять придется все его фрагменты. То же самое и в случае "смещающегося окна" данных, когда старые данные удаляются, а новые — добавляются в таблицу в виде отдельного фрагмента. При этом нет необходимости изменять локально фрагментированные индексы, а вот все глобально фрагментированные придется изменить. В некоторых случаях сервер Oracle может учитывать факт локальной фрагментации индекса аналогично таблице, и вырабатывать на основе этого факта оптимальные планы выполнения запросов. При использовании глобально фрагментированных индексов такой взаимосвязи фрагментов индекса и таблицы нет. Локальные индексы также помогают при восстановлении состояния фрагмента на определенный момент времени. Если необходимо восстановить состояние одного фрагмента на более ранний момент времени, чем всю остальную таблицу, все локально фрагментированные индексы можно восстановить на этот же момент. Все глобально фрагментированные индексы для такого объекта придется пересоздавать.

Выделяется два типа локально фрагментированных индексов.

- **Локально фрагментированные индексы с префиксом.** Это индексы, в которых ключи фрагментации являются начальными ключами индекса. Например, если таблица фрагментирована по диапазону значений столбца **TIMESTAMP,** в списке столбцов ключа локально фрагментированного индекса с префиксом по этой таблице столбец **TIMESTAMP** будет первым.
- **Локально фрагментированные индексы без префикса.** Это индексы, в которых ключи фрагментации **не** являются начальными ключами индекса. Такой индекс может содержать или не содержать столбцы ключа фрагментации.

Оба типа индексов обеспечивают игнорирование фрагмента, оба могут поддерживать уникальность (если индекс без префикса включает ключ фрагментации) и т.д. Запрос, использующий локально фрагментированный индекс с префиксом, всегда **позволяет** пропускать фрагмент, а запрос, использующий локально фрагментированный индекс без префикса, может и не позволить это сделать. Вот почему утверждается, что локально фрагментированные индексы без префикса "медленнее"; они **не гарантируют** игнорирование фрагментов (хотя его и поддерживают). Кроме того, как будет продемонстрировано ниже, при выполнении некоторых операций оптимизатор будет обрабатывать локально фрагментированные индексы без префикса не так, как индексы с префиксом. В документации Oracle подчеркивается, что:

> локально фрагментированные индексы с префиксом обеспечивают более высокую производительность, чем индексы без префикса, потому что уменьшают количество проверяемых оптимизатором индексов

Понимать это надо так:

> локально фрагментированные индексы обеспечивают более высокую производительность для. ЗАПРОСОВ, ссылающихся на весь входящий в них ключ фрагментации, по сравнению с ЗАПРОСАМИ, не ссылающимися на ключ фрагментации

Локально фрагментированные индексы с префиксом, использующиеся для начального доступа к таблице в запросе, не имеют существенных преимуществ по сравнению с индексами без префикса. Я имею в виду, что, если выполнение запроса может начаться с просмотра индекса (**SCAN AN INDEX),** особой разницы между индексами с префиксом и без префикса нет. Позже, когда будет рассматриваться использование фрагментированных индексов в соединениях, вы увидите разницу между индексами с префиксом и без префикса.

Для запроса, выполнение которого начинается с доступа по индексу, все зависит от условия, использованного в запросе. Продемонстрирую это на маленьком примере. Создадим таблицу **PARTITIONED_TABLE** и локально фрагментированный индекс с префиксом **LOCAL_PREFIXED** по ней. Кроме того, добавим локально фрагментированный индекс без префикса **LOCAL_NONPREFIXED:**

```
tkyte@TKYTE816> CREATE TABLE partitioned_table
  2  (a int,
  3   b int
  4  )
  5  PARTITION BY RANGE (a)
  6  (
  7  PARTITION part_1 VALUES LESS THAN(2),
  8  PARTITION part_2 VALUES LESS THAN(3)
  9  )
 10  /

Table created.

tkyte@TKYTE816> create index local_prefixed on partitioned_table (a,b)
  local;

Index created.
```

```
tkyte@TKYTE816> create index local_nonprefixed on partitioned_table (b)
  local;

Index created.
```

Теперь вставим данные в один фрагмент и пометим фрагменты индексов как не используемые (**UNUSABLE**):

```
tkyte@TKYTE816> insert into partitioned_table values (1, 1);

1 row created.

tkyte@TKYTE816> alter index local_prefixed modify partition part_2 unusable;

Index altered.

tkyte@TKYTE816> alter index local_nonprefixed modify partition part_2
  unusable;

Index altered.
```

Пометка этих фрагментов индекса как **UNUSABLE** предотвращает доступ к ним сервера Oracle. Все будет точно так же, как если бы произошел сбой носителя, — фрагменты недоступны. Теперь выполним запросы к таблице, чтобы разобраться, какие фрагменты индексов потребуются для разных запросов:

```
tkyte@TKYTE816> set autotrace on explain
tkyte@TKYTE816> select * from partitioned_table where a = 1 and b = 1;

         A          B

         1          1

Execution Plan

   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=26)
   1    0   INDEX (RANGE SCAN) OF 'LOCAL_PREFIXED' (NON-UNIQUE) (Cost=1
```

Итак, запрос, использующий индекс **LOCAL_PREFIX**, успешно выполнен. Оптимизатор смог исключить фрагмент **PART_2** индекса **LOCAL_PREFIX** из рассмотрения, поскольку в запросе задано условие **A=1**. Нам помогло игнорирование фрагмента. Для второго запроса:

```
tkyte@TKYTE816> select * from partitioned_table where b = 1;
ERROR:
ORA-01502: index 'TKYTE.LOCAL_NONPREFIXED' or partition of such index is
in unusable state

no rows selected

Execution Plan

   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=2 Bytes=52)
   1    0   PARTITION RANGE (ALL)
   2    1     TABLE ACCESS (BY LOCAL INDEX ROWID) OF 'PARTITIONED_TABLE'
   3    2       INDEX (RANGE SCAN) OF 'LOCAL_NONPREFIXED' (NON-UNIQUE)
```

Оптимизатор **не смог** исключить из рассмотрения фрагмент **PART_2** индекса **LOCAL_NONPREFIXED.** С этим и связана проблема производительности при исполь-

зовании локально фрагментированных индексов без префикса. Они используются и для запросов, **не включающих ключ фрагментации,** в отличие от индексов с префиксом. Дело не в том, что индексы с префиксом лучше, просто они используются запросами, обеспечивающими возможность игнорирования фрагментов.

Если удалить индекс **LOCAL_PREFIXED** и еще раз выполнить исходный, успешно выполненный запрос:

```
tkyte@TKYTE816> select * from partitioned_table where a = 1 and b = 1;

         A          B

         1          1

Execution Plan

   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=26)
   1    0   TABLE ACCESS (BY LOCAL INDEX ROWID) OF 'PARTITIONED_TABLE'
   2    1     INDEX (RANGE SCAN) OF 'LOCAL_NONPREFIXED' (NON-UNIQUE) (Cost=1
```

Этот результат может показаться удивительным. Почти такой же план выполнения, как и в случае неудавшегося запроса, но на этот раз все работает. Причина в том, что оптимизатор может пропускать фрагменты даже для локально фрагментированных индексов без префикса (в этом плане нет шага PARTITION RANGE(ALL)).

Если к представленной выше таблице часто выполняются запросы вида:

```
select ... from partitioned_table where a = :a and b = :b;
select ... from partitioned_table where b = :b;
```

имеет смысл создать локально фрагментированный индекс без префикса по столбцам **(b,a)**; он пригодится для обоих представленных выше запросов. Локально фрагментированный индекс с префиксом по столбцам **(a,b)** пригодится только для первого запроса.

Однако при использовании фрагментированных индексов в соединениях результаты могут быть другими. В представленных выше примерах сервер Oracle по условию запроса мог во время оптимизации определить, можно или нельзя пропустить фрагменты. Это было понятно по условию в конструкции **WHERE** (даже если в нем использовались связываемые переменные). Когда доступ по индексу используется в качестве начального, основного метода доступа, особой разницы между локально фрагментированными индексами с префиксом и без префикса нет. При соединении с локально фрагментированным индексом, однако, все меняется. Рассмотрим следующую таблицу, фрагментированную по диапазону:

```
tkyte@TKYTE816> CREATE TABLE range_example
  2  (range_key_column date,
  3   x              int,
  4   data           varchar2(20)
  5  )
  6  PARTITION BY RANGE (range_key_column)
  7  (PARTITION part_1 VALUES LESS THAN
  8     (to_date('01-jan-1995',     'dd-mon-yyyy')),
  9   PARTITION part_2 VALUES LESS THAN
 10        (to_date('01-jan-1996','dd-mon-yyyy'))
 11  )
```

```
 12  /

Table created.

tkyte@TKYTE816> alter table range_example
  2  add constraint range_example_pk
  3  primary key (range_key_column,x)
  4  using index local
  5  /

Table altered.

tkyte@TKYTE816> insert into range_example values (to_date('01-jan-1994'), 1,
'xxx');

1 row created.

tkyte@TKYTE816> insert into range_example values (to_date('01-jan-1995'), 2,
  'xxx');

1 row created.
```

Сначала по таблице создан локально фрагментированный индекс с префиксом для первичного ключа. Чтобы увидеть разницу между индексами с префиксом и без префикса, необходимо создать еще одну таблицу. Используем эту таблицу в качестве ведущей в запросе к таблице **RANGE_EXAMPLE**. Таблица **TEST** просто используется в качестве ведущей в запросе, который будет выполнять соединение вложенными циклами с таблицей **RANGE_EXAMPLE**:

```
tkyte@TKYTE816> create table test (pk, range_key_column, x,
  2                        constraint test_pk primary key(pk))
  3  as
  4  select rownum, range_key_column, x from range_example
  5  /

Table created.

tkyte@TKYTE816> set autotrace on explain

tkyte@TKYTE816> select * from test, range_example
  2    where test.pk - 1
  3     and test.range_key_column = range_example.range_key_column
  4     and test.x = range_example.x
  5  /
```

PK	RANGE_KEY	X	RANGE_KEY	X	DATA
1	01-JAN-94	1	01-JAN-94	1	xxx

Execution Plan

```
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=69)
1    0   NESTED LOOPS (Cost=2 Card=1 Bytes=69)
2    1     TABLE ACCESS (BY INDEX ROWID) OF 'TEST' (Cost=1 Card=1
3    2       INDEX (RANGE SCAN) OF 'TEST_PK' (UNIQUE) (Cost=1 Card=1)
4    1     PARTITION RANGE (ITERATOR)
5    4       TABLE ACCESS (BY LOCAL INDEX ROWID) OF 'RANGE_EXAMPLE'
6    5         INDEX (UNIQUE SCAN) OF 'RANGE_EXAMPLE_PK' (UNIQUE)
```

Представленный выше план будет обрабатываться следующим образом.

1. С помощью индекса **TEST_PK** находим строки в таблице **TEST**, соответствующие условию **test.pk = 1.**
2. Обращаемся к таблице **TEST** для выборки значений остальных столбцов **TEST** — **range_key_column** и **x**.
3. По выбранным на предыдущем шаге значениям с помощью **RANGE_EXAMPLE_PK** находим единственный соответствующий фрагмент со строками таблицы **RANGE_EXAMPLE**.
4. Обращаемся к одному фрагменту таблицы **RANGE_EXAMPLE** для выбора значений столбца данных.

Это кажется достаточно очевидным, но давайте посмотрим, что произойдет при изменении порядка следования столбцов **range_key_column** и **x** и превращении индекса в индекс без префикса:

```
tkyte@TKYTE816> alter table range_example
  2  drop constraint range_example_pk
  3  /

Table altered.

tkyte@TKYTE816> alter table range_example
  2  add constraint range_example_pk
  3  primary key (x,range_key_column)
  4  using index local
  5  /

Table altered.

tkyte@TKYTE816> select * from test, range_example
  2    where test.pk = 1
  3    and test.range_key_column = range_example.range_key_column
  4    and test.x = range_example.x
  5  /

        PK RANGE_KEY            X       RANGE_KEY       X DATA

         1 01-JAN-94            1  01-JAN-94            1 xxx

Execution Plan

   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=69)
   1    0   NESTED LOOPS (Cost=2 Card=1 Bytes=69)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'TEST' (Cost=1 Card=1
   3    2       INDEX (RANGE SCAN) OF 'TEST_PK' (UNIQUE) (Cost=1 Card=1)
   4    1     PARTITION RANGE (ITERATOR)
   5    4       TABLE ACCESS (FULL) OF 'RANGE_EXAMPLE' (Cost=1 Card=164
```

Неожиданно оказывается, что с точки зрения сервера Oracle этот новый индекс слишком неэффективен. Это один из случаев, когда использование индекса с префиксом гораздо предпочтительнее.

Итак, не стоит бояться локально фрагментированных индексов без префикса или считать их причиной снижения производительности. Если имеется много запросов, которые могут использовать индекс без префикса, как было показано выше, имеет смысл его создать. Главное — проверить, есть ли в запросах условия, позволяющие использовать игнорирование фрагментов индекса. При использовании локально фрагментированных индексов с префиксом это гарантируется. При использовании индексов без префикса — нет. Следует также учитывать, как именно используется индекс: при использовании этих двух типов индексов на первом шаге выполнения запроса особых различий нет. Если же индекс используется для выполнения соединения, как в предыдущем примере, индексы с префиксом имеют преимущество. Если можно использовать локально фрагментированный индекс, создавайте именно его.

Локально фрагментированные индексы и уникальность

Для поддержки уникальности, задаваемой требованиями целостности UNIQUE или PRIMARY KEY, ключ фрагментации индекса должен входить в соответствующее требование. По-моему, это самая главная особенность локально фрагментированных индексов. Сервер Oracle обеспечивает уникальность только в пределах фрагмента индекса, но не среди нескольких фрагментов. Это означает, например, что нельзя фрагментировать данные по диапазону значений столбца TIMESTAMP и обеспечить поддержку первичного ключа по столбцу ID с помощью локально фрагментированного индекса. Сервер Oracle для обеспечения уникальности создаст один глобальный индекс.

Например, если выполнить следующий оператор CREATE TABLE в схеме, где нет объектов (чтобы можно было легко понять, какие объекты созданы, просто запросив все сегменты данного пользователя), окажется:

```
tkyte@TKYTE816> CREATE TABLE partitioned
  2  (timestamp date,
  3   id       int primary key
  4  )
  5  PARTITION BY RANGE (timestamp)
  6  (
  7  PARTITION part_1 VALUES LESS THAN
  8   (to_date('01-jan-2000','dd-mon-yyyy')),
  9  PARTITION part_2 VALUES LESS THAN
 10 (to_date('01-jan-2001','dd-mon-yyyy'))
 11  )
 12  /

Table created.

tkyte@TKYTE816> select segment_name, partition_name, segment_type
  2    from user_segments;

SEGMENT_NAME     PARTITION_NAME    SEGMENT_TYPE

PARTITIONED      PART_2            TABLE PARTITION
PARTITIONED      PART_1            TABLE PARTITION
SYS_C003582                         INDEX
```

Индекс **SYS_C003582** — нефрагментированный, он и не мог таким оказаться. Это означает, что вы теряете ценные для хранилищ данных свойства фрагментации. Обыч-

ные действия с фрагментами в хранилищах данных больше нельзя будет выполнять независимо. Так, при добавлении нового фрагмента данных, придется пересоздавать весь глобальный индекс, а также изменять все действия с фрагментами. Сравните с таблицей, имеющей только локально фрагментированные индексы — их не нужно пересоздавать, если только не затрагивается соответствующий фрагмент.

Если попытаться обмануть сервер Oracle, воспользовавшись тем, что требование первичного ключа может обеспечиваться и неуникальным индексом, окажется, что это тоже не помогает:

```
tkyte@TKYTE816> CREATE TABLE partitioned
  2  (timestamp date,
  3   id       int
  4  )
  5  PARTITION BY RANGE (timestamp)
  6  (
  7  PARTITION part_1 VALUES LESS THAN
  8   (to_date('01-jan-2000','dd-mon-yyyy')),
  9  PARTITION part_2 VALUES LESS THAN
 10   (to_date('01-jan-2001','dd-mon-yyyy'))
 11  )
 12  /

Table created.

tkyte@TKYTE816> create index partitioned_index
  2  on partitioned(id)
  3  LOCAL
  4  /
Index created.

tkyte@TKYTE816> select segment_name, partition_name, segment_type
  2      from user_segments;

SEGMENT_NAME          PARTITION_NAME     SEGMENT_TYPE

PARTITIONED           PART_2             TABLE PARTITION
PARTITIONED           PART_1             TABLE PARTITION
PARTITIONED_INDEX     PART_2             INDEX PARTITION
PARTITIONED_INDEX     PART_1             INDEX PARTITION

tkyte@TKYTE816> alter table partitioned
  2  add constraint partitioned_pk.
  3  primary key(id)
  4  /
alter table partitioned

ERROR at line 1:
ORA-01408: such column list already indexed
```

Здесь сервер Oracle попытался создать глобальный индекс по столбцу **ID**, но обнаружил, что не может этого сделать, потому что индекс уже существует. Представленные выше операторы сработали бы, если бы был создан нефрагментированный индекс (сервер Oracle просто использовал бы его для выполнения требования).

Если ключ фрагментации не входит в условие, обеспечить уникальность нельзя по двум причинам. Во-первых, если бы сервер Oracle это разрешал, то были бы сведены на нет преимущества фрагментации. Доступность и масштабируемость были бы потеряны, поскольку требовался бы **просмотр** всех фрагментов и доступ к ним при любой вставке или изменении данных. Чем больше фрагментов, тем менее доступны данные. Чем больше фрагментов в таблице, тем больше фрагментов индекса приходится просматривать и тем менее масштабируемой становится схема фрагментации. То есть фрагментация ухудшит оба показателя.

Кроме того, серверу Oracle пришлось бы последовательно выполнять вставки и изменения этой таблицы на уровне транзакций. Дело в том, что при добавлении строки со значением **ID=1** во фрагмент **PART_1,** серверу пришлось бы **предотвращать** вставку строки со значением **ID=1** другими сеансами во фрагмент **PART_2.** Единственный способ добиться этого — запретить изменять фрагмент **PART_2,** поскольку другого метода заблокировать этот фрагмент просто нет.

В системе ООТ для обеспечения целостности данных требования уникальности должны обеспечиваться системой (сервером Oracle). Это означает, что логическая модель данных будет влиять на физическую. Необходимость поддерживать уникальность либо будет определять схему фрагментации, диктуя выбор ключей фрагментации, либо наличие этих требований приведет к необходимости использования **глобально** фрагментированных индексов. Рассмотрим глобально фрагментированные индексы более подробно.

Глобально фрагментированные индексы

Глобально фрагментированные индексы разбиваются на фрагменты не так, как базовая таблица. Таблица может быть разбита по значению столбца **TIMESTAMP** на десять фрагментов, а глобально фрагментированный индекс по этой таблице может быть разбит на пять фрагментов по значению столбца **REGION**. В отличие от локально фрагментированных, есть только один класс глобально фрагментированных индексов — с **префиксом.** Глобально фрагментированные индексы, ключ которых не начинается с ключа фрагментации, не поддерживаются.

Продолжая предыдущий пример, ниже я представлю простой вариант использования глобально фрагментированного индекса. Вы убедитесь, что глобально фрагментированный индекс обеспечивает уникальность первичного ключа, так что можно использовать фрагментированные индексы, обеспечивающие уникальность, но не включающие ключ фрагментации базовой таблицы. В следующем примере создается таблица, фрагментированная по столбцу **TIMESTAMP,** индекс которой фрагментирован по столбцу ID:

```
tkyte@TKYTE816> CREATE TABLE partitioned
  2  (timestamp date,
  3   id        int
  4  )
  5  PARTITION BY RANGE (timestamp)
  6  (
  7  PARTITION part_1 VALUES LESS THAN
  8   (to_date('01-jan-2000','dd-mon-yyyy')),
```

```
  9  PARTITION part_2 VALUES LESS THAN
 10 (to_date('01-jan-2001','dd-mon-yyyy'))
 11  )
 12  /
```

Table created.

```
tkyte@TKYTE816> create index partitioned_index
  2  on partitioned(id)
  3  GLOBAL
  4  partition by range(id)
  5   (
  6  partition part_1 values less than(1000),
  7  partition part_2 values less than (MAXVALUE)
  8   )
  9   /
Index created.
```

Обратите внимание на использование в этом индексе значения **MAXVALUE**. Значение **MAXVALUE** можно использовать в таблицах или индексах, фрагментированных по диапазону. Оно представляет максимально возможное значение. До сих пор в примерах использовались жесткие верхние границы диапазонов (значения, меньшие чем Определенное значение>). Однако для глобально фрагментированного индекса требуется, чтобы фрагмент с наибольшими значениями (последний фрагмент) имел верхний предел **MAXVALUE**. Это гарантирует, что все строки базовой таблицы можно будет проиндексировать.

Добавим для таблицы первичный ключ:

```
tkyte@TKYTE816> alter table partitioned add constraint
  2  partitioned_pk
  3  primary key(id)
  4  /

Table altered.
```

Пока еще не очевидно, что сервер Oracle использует для поддержки первичного ключа созданный индекс. Доказать это можно с помощью "волшебного" запроса к словарю данных. Этот запрос необходимо выполнять от имени учетной записи, имеющей привилегию **SELECT** на базовые таблицы словаря данных или привилегию **SELECT ANY TABLE**:

```
tkyte@TKYTE816> select  t.name    table_name
  2       ,  u.name    owner
  3       ,   c.name   constraint_name
  4       ,  i.name    index_name
  5       ,   decode(bitand(i.flags, 4), 4, 'Yes',
  6              decode(i.name, c.name, 'Possibly', 'No')) generated
  7   from sys.cdef$ cd
  8      , sys.con$  c
  9      ,  sys.obj$  t
 10      , sys.obj$  i
 11      , sys.user$ u
 12  where   cd.type#   between 2 and 3
 13    and     cd.con#     = c.con#
```

```
 14    and      cd.obj#    = t.obj#
 15    and     cd.enabled = i.obj#
 16    and      c.owner#       = u.user#
 17    and   c.owner#    = uid
 18  /

TABLE_NAME   OWNER CONSTRAINT_NAME INDEX_NAME       GENERATE

PARTITIONED  TKYTE PARTITIONED_PK  PARTITIONED_INDEX No
```

Запрос показывает, какой индекс использовался для поддержки данного требования, и пытается "угадать", сгенерировано ли **имя** индекса автоматически **или** задано явно. В данном случае он показывает, что для поддержки первичного ключа используется только что созданный индекс **PARTITIONED_INDEX** (имя которого не было сгенерировано автоматически).

Чтобы показать, что сервер Oracle не позволит создать глобальный индекс **без** префикса, достаточно попробовать:

```
tkyte@TKYTE816> create index partitioned_index2
  2  on partitioned(timestamp,id)
  3  GLOBAL
  4  partition  by range(id)
  5   (
  6  partition part_1 values less than(1000),
  7  partition part_2 values less than(MAXVALUE)
  8   )
  9   /
partition  by range(id)
                  *
ERROR at line 4:
ORA-14038: GLOBAL partitioned index must be prefixed
```

Сообщение об ошибке весьма красноречиво. Глобально фрагментированный индекс **должен** быть с префиксом.

Итак, когда же надо использовать глобально фрагментированный индекс? Рассмотрим два типа систем — хранилища данных и системы OOT — и выясним, когда эти индексы могут пригодиться.

Хранилища данных и глобально фрагментированные индексы

Я считаю, что эти две вещи несовместимы. Хранилища данных предполагают определенные особенности; добавляются и удаляются большие объемы данных, высока вероятность сбоя на каком-нибудь из дисков и т.д. В любом хранилище данных, использующем перемещающееся окно, лучше избегать использования глобально фрагментированных индексов. Вот пример того, что я имею в виду под перемещающимся окном, и как на его использование влияет глобально фрагментированный индекс:

```
tkyte@TKYTE816>CREATE TABLE partitioned
  2   (timestamp date,
  3    id       int
  4   )
  5  PARTITION BY RANGE (timestamp)
  6   (
```

```
  7  PARTITION fy_1999 VALUES LESS THAN
  8  (to_date('01-jan-2000','dd-mon-yyyy')),
  9  PARTITION fy_2000 VALUES LESS THAN
 10  (to_date('Ol-jan-2001','dd-mon-yyyy')),
 11  PARTITION the_rest VALUES LESS THAN
 12  (maxvalue)
 13  )
 14  /

Table created.

tkyte@TKYTE816> insert into partitioned partition(fy_1999)
  2  select to_date('31-dec-1999')-mod(rownum,360), object_id
  3  from all_objects
  4  /

21914 rows created.

tkyte@TKYTE816> insert into partitioned partition(fy_2000)
  2  select to_date('31-dec-2000')-mod(rownum,360), object_id
  3  from all_objects
  4  /

21914 rows created.

tkyte@TKYTE816> create index partitioned_idx_local
  2  on partitioned(id)
  3  LOCAL
  4  /

Index created.

tkyte@TKYTE816> create index partitioned_idx_global
  2  on partitioned(timestamp)
  3  GLOBAL
  4  /

Index created.
```

Итак, мы создали таблицу "хранилища данных". Данные фрагментированы по финансовому году; оперативно доступны данные за последние два года. По этой таблице создано два фрагментированных индекса; один — как **LOCAL**, а другой — как **GLOBAL**. Обратите внимание, что я оставил пустой фрагмент, **THE_REST**, в конце таблицы. Это поможет быстро добавлять новые данные. Предположим, очередной финансовый год закончился, и необходимо сделать следующее.

1. Удалить данные за самый давний финансовый год. Эти данные не теряются, они просто считаются устаревшими и архивируются.
2. Добавить данные за последний финансовый год. Для их загрузки, преобразования, индексирования и т.д. потребуется определенное время. Хотелось бы, чтобы эти действия не влияли на доступность остальных данных.

Итак, можно выполнить следующее:

```
tkyte@TKYTE816> create table fy_1999 (timestamp date, id int);

Table created.
```

```
tkyte@TKYTE816> create index fy_1999_idx on fy_1999(id)
  2  /

Index created.

tkyte@TKYTE816> create table fy_2001 (timestamp date, id int);

Table created.

tkyte@TKYTE816> insert into fy_2001
  2  select to_date('31-dec-2001')-mod(rownum,360), object_id
  3   from all_objects
  4  /

21922 rows created.

tkyte@TKYTE816> create index fy_2001_idx on fy_2001(id) nologging
  2  /

Index created.
```

Здесь я создал новую пустую таблицу-"оболочку" и индекс для самых старых данных. Преобразуем текущий полный фрагмент в пустой и создадим "полную" таблицу с данными фрагмента **FY_1999**. Кроме того, я заранее выполнил все необходимые действия по подготовке данных для фрагмента **FY_2001**. Речь идет о проверке достоверности данных, преобразовании и любых других сложных действиях по их подготовке.

Теперь все готово для изменения "актуальных" данных:

```
tkyte@TKYTE816> alter table partitioned
  2  exchange partition fy_1999
  3  with table fy_1999
  4   including indexes
  5  without validation
  6  /

Table altered.

tkyte@TKYTE816> alter table partitioned
  2  drop partition fy_1999
  3  /

Table altered.
```

Вот и все, что необходимо сделать для удаления "устаревших" данных. Мы превратили фрагмент в отдельную целую таблицу, а пустую таблицу — во фрагмент. Это было просто изменение словаря данных, никаких больших объемов ввода-вывода. Теперь можно экспортировать эту таблицу (возможно, с помощью перемещаемого табличного пространства) из базы данных с целью архивирования. При необходимости ее очень легко присоединить снова.

Теперь добавим новые данные:

```
tkyte@TKYTE816> alter table partitioned
  2  split partition the_rest
  3  at  (to_date('01-jan-2002','dd-mon-yyyy'))
  4  into (partition fy_2001, partition the_rest)
  5  /

Table altered.
```

```
tkyte@TKYTE816> alter table partitioned
  2 exchange partition fy_2001
  3  with table fy_2001
  4  including indexes
  5 without validation
  6  /

Table altered.
```

Изменение словаря данных было выполнено моментально. Отделение пустого фрагмента требует очень мало времени, поскольку данных там никогда не было и не будет. Вот зачем я поместил пустой фрагмент в конец таблицы, — чтобы упросить отделение. Затем вновь созданный пустой фрагмент заменяется всей таблицей, а таблица — пустым фрагментом. Новые данные оперативно доступны.

Однако, если посмотреть на индексы, оказывается:

```
tkyte@TKYTE816> select index_name, status from user_indexes;

INDEX_NAME                     STATUS

FY_1999_IDX                    VALID
FY_2001_IDX                    VALID
PARTITIONED_IDX_GLOBAL         UNUSABLE
PARTITIONED_IDX_LOCAL          N/A
```

Глобально фрагментированный индекс после этих действий, несомненно, недоступен. Поскольку каждый фрагмент индекса может указывать на фрагмент таблицы, и мы удалили один фрагмент, и добавили другой, индекс некорректен. В нем есть записи, ссылающиеся на удаленный фрагмент. В нем **нет** записей, ссылающихся на добавленный фрагмент. Запрос, использующий этот индекс, не будет выполнен:

```
tkyte@TKYTE816> select count(*)
  2  from partitioned
  3  where timestamp between sysdate-50 and sysdate;
select count(*)
*
ERROR at line 1:
ORA-01502: index 'TKYTE.PARTITIONED_IDX_GLOBAL' or partition of such
index is in unusable state
```

Можно установить параметр **SKIP_UNUSABLE_INDEXES=TRUE**, но тогда мы теряем повышение производительности, которое обеспечивал индекс (это работает в Oracle 8.1.5 и более поздних версиях; до этого оператор **SELECT** все равно пытался бы использовать индекс, помеченный как **UNUSABLE**). Этот индекс надо пересоздать, чтобы снова обеспечить возможность использования данных. Процесс перемещения окна, который до сих пор происходил без задержек, теперь будет выполняться очень долго, пока не будет пересоздан глобальный индекс. Придется просмотреть все данные и полностью пересоздать индекс по данным таблицы. Если таблица имеет размер в сотни гигабайт, для этого потребуются существенные ресурсы.

Любая операция с фрагментом таблицы сделает невозможным использование глобально фрагментированного индекса. Если необходимо перенести фрагмент на другой диск, все глобально фрагментированные индексы надо пересоздавать (для локально фрагмен-

тированных индексов достаточно пересоздать только **соответствующие** фрагменты). Если окажется, что необходимо разделить фрагмент на два меньших, все глобально фрагментированные индексы придется пересоздавать (для локально фрагментированных индексов достаточно пересоздать только соответствующие пары фрагментов). И так далее. Поэтому надо избегать использования глобально фрагментированных индексов в среде хранилища данных. Их использование может негативно повлиять на многие действия.

Системы OOT и глобально фрагментированные индексы

Система OOT характеризуется частым выполнением множества небольших транзакций, читающих и изменяющих данные. Обычно беспокоиться о поддержке перемещающихся окон данных не приходится. Прежде всего необходимо обеспечить быстрый доступ к строкам. Целостность данных — жизненно важна. Доступность данных также имеет большое значение.

Глобально фрагментированные индексы имеет смысл использовать в системах OOT. Данные таблицы могут быть фрагментированы только по одному ключу, по одному набору столбцов. Однако могут понадобиться различные способы доступа к данным. Можно фрагментировать данные в таблице **EMPLOYEE** по местонахождению офиса. Однако при этом необходимо также обеспечить быстрый доступ к данным по следующим столбцам.

- **DEPARTMENT.** Отделы географически разнесены, и однозначного соответствия между отделом и его местонахождением нет.
- **EMPLOYEE_ID.** Хотя идентификатор сотрудника определяет его местонахождение, не хотелось бы искать данные по **EMPLOYEE_ID** и **LOCATION,** поскольку при этом не удастся обеспечить игнорирование фрагментов индекса. Кроме того, значения столбца **EMPLOYEE_ID** сами по себе должны быть **уникальны.**
- **JOB_TITLE.**

Необходимо обращаться к данным таблицы **EMPLOYEE** по многим различным ключам, из разных частей приложения, причем, скорость доступа является основным требованием. В хранилище данных мы просто использовали бы локально фрагментированные индексы по перечисленным выше ключам и параллельный просмотр диапазонов по индексам для быстрого доступа к данным. Там не нужно было бы использовать игнорирование фрагментов, но в системе OOT, однако, это необходимо. Распараллеливание запроса в таких системах неприемлемо — надо предоставить соответствующие индексы. Поэтому необходимо использовать глобально фрагментированные индексы по некоторым полям.

Итак, необходимо достичь следующих целей:

- быстрый доступ;
- целостность данных;
- доступность данных.

В системе **OOT** этого позволяют добиться глобально фрагментированные индексы, поскольку характеристики этой системы существенно отличаются от хранилища данных. Не будут использоваться перемещающиеся окна, не придется делить фрагменты (разве

что в период запланированного простоя), не нужно переносить данные из одного табличного пространства в другое и т.д. Действия, типичные для хранилищ данных, обычно не выполняются в системе оперативной обработки транзакций.

Вот небольшой пример, показывающий, как добиться трех перечисленных выше целей с помощью глобально фрагментированных индексов. Я собираюсь использовать простые глобальные индексы из одного фрагмента, но результаты будут такими же и для глобально фрагментированных индексов (разве что **возрастет** доступность и управляемость при добавлении фрагментов):

```
tkyte@TKYTE816> create table emp
  2   (EMPNO             NUMBER(4) NOT NOLL,
  3    ENAME             VARCHAR2(10),
  4    JOB               VARCHAR2(9),
  5    MGR               NUMBER(4),
  6    HIREDATE          DATE,
  7    SAL               NUMBER(7,2),
  8    COMM              NUMBER(7,2),
  9    DEPTNO            NUMBER(2) NOT NULL,
 10    LOC               VARCHAR2 (13) NOT NULL
 11   )
 12  partition by range(loc)
 13   (
 14  partition p1 values less than('C') tablespace p1,
 15  partition p2 values less than('D') tablespace p2,
 16  partition p3 values less than('N') tablespace p3,
 17  partition p4 values less than('Z') tablespace p4
 18   )
 19   /

Table created.

tkyte@TKYTE816> alter table emp add constraint emp_pk
  2  primary key(empno)
  3  /

Table altered.

tkyte@TKYTE816> create index emp_job_idx on emp(job)
  2  GLOBAL
  3  /

Index created.

tkyte@TKYTE816> create index emp_dept_idx on emp(deptno)
  2  GLOBAL
  3  /

Index created.

tkyte@TKYTE816> insert into emp
  2  select e.*, d.loc
  3    from scott.emp e, scott.dept d
  4   where e.deptno = d.deptno
  5  /

14 rows created.
```

Итак, мы начинаем с таблицы, фрагментированной по местонахождению, **LOC**, в соответствии с нашими правилами. Существует глобальный уникальный индекс по столбцу **EMPNO** как побочный эффект выполнения оператора **ALTER TABLE ADD CONSTRAINT.** Это показывает, что можно обеспечить целостность данных. Кроме того, мы добавили еще два глобальных индекса по столбцам **DEPTNO** и **JOB** для ускорения доступа к строкам по этим атрибутам. Теперь добавим в таблицу немного данных и посмотрим, что оказалось в каждом из фрагментов:

```
tkyte@TKYTE816> select empno,job,loc from emp partition(p1);

no rows selected

tkyte@TKYTE816> select empno,job,loc from emp partition(p2);

     EMPNO  JOB       LOC

      7900  CLERK     CHICAGO
      7844  SALESMAN  CHICAGO
      7698  MANAGER   CHICAGO
      7654  SALESMAN  CHICAGO
      7521  SALESMAN  CHICAGO
      7499  SALESMAN  CHICAGO
6 rows selected.

tkyte@TKYTE816> select empno,job,loc from emp partition(p3);

     EMPNO  JOB       LOC

      7902  ANALYST   DALLAS
      7876  CLERK     DALLAS
      7788  ANALYST   DALLAS
      7566  MANAGER   DALLAS
      7369  CLERK     DALLAS

tkyte@TKYTE816> select empno,job,loc from emp partition(p4);

     EMPNO  JOB       LOC

      7934  CLERK     NEW YORK
      7839  PRESIDENT NEW YORK
      7782  MANAGER   NEW YORK
```

Этот пример показывает распределение данных по фрагментам в соответствии с местонахождением сотрудника. Теперь можно выполнить несколько запросов для оценки производительности:

```
tkyte@TKYTE816> select empno,job,loc from emp where empno = 7782;

     EMPNO  JOB       LOC

      7782  MANAGER   NEW YORK

Execution Plan

   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=4 Bytes=108)
   1    0   TABLE ACCESS (BY GLOBAL INDEX ROWID) OF 'EMP' (Cost=1 Card
   2    1     INDEX (RANGE SCAN) OF 'EMP_PK' (UNIQUE) (Cost=1 Card=4)
```

```
tkyte@TKYTE816> select empno,job,loc from emp where job = 'CLERK';

    EMPNO  JOB        LOC

     7900  CLERK      CHICAGO
     7876  CLERK      DALLAS
     7369  CLERK      DALLAS
     7934  CLERK      NEW YORK

Execution Plan

   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=4 Bytes=108)
   1    0   TABLE ACCESS (BY GLOBAL INDEX ROWID) OF 'EMP' (Cost=1 Card
   2    1     INDEX (RANGE SCAN) OF 'EMP_JOB_IDX' (NON-UNIQUE) (Cost=1
```

Созданные индексы используются для обеспечения высокоскоростного доступа к данным в системе OOT. Если бы они были фрагментированы, то должны были бы включать префикс, что позволило бы игнорировать фрагменты индекса; но и так масштабируемость вполне приемлема. Наконец, давайте разберемся **с** доступностью данных. Документация Oracle утверждает, что глобально фрагментированные индексы обеспечивают "меньшую доступность" данных, чем локально фрагментированные. Не могу полностью согласиться с этой безоговорочной характеристикой. Я уверен, что в системе OOT **они обеспечивают такую же степень доступности, как и локально фрагментированные.** Рассмотрим пример:

```
tkyte@TKYTE816> alter tablespace p1 offline;

Tablespace altered.

tkyte@TKYTE816> alter tablespace p2 offline;

Tablespace altered.

tkyte@TKYTE816> alter tablespace p3 offline;

Tablespace altered.

tkyte@TKYTE816> select empno,job,loc from emp where empno = 7782;

    EMPNO JOB        LOC

     7782 MANAGER    NEW YORK

Execution Plan

   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=4 Bytes=108)
   1    0   TABLE ACCESS (BY GLOBAL INDEX ROWID) OF 'EMP' (Cost=1 Card
   2    1     INDEX (RANGE SCAN) OF 'EMP PK' (UNIQUE) (Cost=1 Card=4)
```

Итак, хотя большая часть базовых данных таблицы недоступна, мы можем добраться до любого компонента данных по индексу. Если только необходимое значение **EMPNO** находится в доступном табличном пространстве, глобальный индекс используется. С другой стороны, если бы нас **угораздило** использовать в этом случае "обеспечивающий высокую доступность данных" локально фрагментированный индекс, это могло бы помешать доступу к данным! Это побочный эффект того, что фрагментация выполнена по столбцу **LOC**, а в запросе задано условие по столбцу **EMPNO**; нам пришлось бы обра-

титься к каждому фрагменту локально фрагментированного индекса, и при попытке обратиться к недоступному фрагменту произошла бы ошибка.

Другие типы запросов не будут (и не могут) нормально выполняться в такой ситуации:

```
tkyte@TKYTE816> select empno,job,loc from emp where job = 'CLERK';
select empno,job,loc from emp where job = 'CLERK'
                  *
ERROR at line 1:
ORA-00376: file 13 cannot be read at this time
ORA-01110: data file 13: 'C:\ORACLE\ORADATA\TKYTE816\P2.DBF'
```

Данные о клерках разбросаны по всем фрагментам; то, что три табличных пространства недоступны, не будет учтено. Это неизбежно, если только данные не фрагментированы по столбцу **JOB**, но тогда проблемы возникли бы с запросами, обращающимися к данным по значению столбца **LOC**. Каждый раз, когда необходимо обращаться к данным по нескольким различным ключам, возникают подобные проблемы. Сервер Oracle предоставит данные, если только это вообще возможно.

Учтите, однако, что если ответ на запрос можно было бы получить по индексу, избежав доступа к таблице **TABLE ACCESS BY ROWID**, фактор недоступности данных не был бы столь критичным:

```
tkyte@TKYTE816> select count(*) from emp where job = 'CLERK';

  COUNT(*)

         4

Execution Plan

    0     SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=6)
    1    0   SORT (AGGREGATE)
    2    1     INDEX (RANGE SCAN) OF 'EMP_JOB_IDX' (NON-UNIQUE) (Cost=1
```

Поскольку серверу Oracle в этом случае таблица не нужна, недоступность большинства фрагментов не скажется на выполнении запроса. Поскольку такая оптимизация (ответ на запрос с помощью одного только индекса) типична для систем OOT, многие приложения "не заметят" недоступности данных. Осталось только как можно быстрее обеспечить доступность этих данных (путем восстановления и синхронизации).

Резюме

Фрагментация особенно полезна как средство повышения масштабируемости при увеличении размеров больших объектов в базе данных. Повышение же масштабируемости положительно сказывается на производительности, доступности данных и упрощает администрирование. Все три последствия крайне важны для разных категорий пользователей. Для администратора базы данных имеет значение возможность эффективного управления. Владельцев системы интересует доступность данных. Простой — это потеря денег, и все, что сокращает простой (или минимизирует его влияние), повышает отдачу

от системы. Пользователей системы интересует производительность, медленно работающие системы никто не любит.

Мы также выяснили, что в системе ООТ фрагментация может и не повысить производительность, особенно при неправильной реализации. Фрагментация повышает производительность выполнения тех классов запросов, которые нехарактерны для систем ООТ. Это важно понимать, поскольку многие считают фрагментацию средством "безусловного повышения производительности". Это не означает, что фрагментацию **не надо** использовать в системах ООТ — она обеспечивает в этой среде другие, менее заметные преимущества. Сокращается время простоев. Производительность остается удовлетворительной (фрагментация при правильном применении не замедлит работу). Упрощается управление системой, вследствие чего повышается производительность, поскольку некоторые действия по сопровождению администратор базы данных выполняет чаще — они ведь выполняются быстрее.

Мы изучили различные схемы фрагментации таблиц, предлагаемые сервером: по диапазону, по хеш-функции и смешанную фрагментацию, и обсудили, для каких случаев каждая из них больше всего подходит. Существенное внимание было уделено фрагментации индексов, оценке различий между индексами с префиксом и без префикса, локально и глобально фрагментированными. Оказалось, что глобально фрагментированные индексы не подходят для большинства хранилищ данных, но в системе **ООТ** именно они используются чаше всего.

Предоставляемые СУБД Oracle возможности фрагментации постоянно развиваются, причем, на следующие версии запланированы существенные улучшения. Со временем, вследствие увеличения размеров баз данных и сложности приложений, фрагментация будет, как мне кажется, использоваться еще более широко. Сеть Internet и присущие ей аппетиты в отношении баз данных приводят к созданию все больших подборок данных, а фрагментация является естественным средством, позволяющим справиться с возникающими при этом проблемами.

15

Автономные транзакции

Автономные транзакции позволяют создать новую транзакцию в пределах текущей, так что можно фиксировать или откатывать ее изменения независимо от родительской транзакции. Они позволяют приостановить текущую транзакцию, начать новую, выполнить ряд действий, зафиксировать их или откатить, не влияя на состояние текущей транзакции. Автономные транзакции предлагают новый метод управления транзакциями в языке PL/SQL и могут использоваться:

- в анонимных блоках верхнего уровня;
- в локальных, отдельных или входящих в пакеты процедурах и функциях;
- в методах объектных типов;
- в триггерах базы данных.

Для выполнения примеров, приводимых в этой главе, необходим сервер Oracle версии 8.1.5 или выше. Подходит любая редакция — Standard, Enterprise или Personal, поскольку эта возможность поддерживается во всех редакциях.

В этой главе мы:

- выясним, для чего используются автономные транзакции, включая реализацию проверки, записи которой нельзя откатить, предотвращение возникновения ошибок изменяющихся таблиц, запись в базу данных из оператора **SELECT** и повышение модульности кода;
- рассмотрим, как работают автономные транзакции; изучим управление транзакциями и область действия транзакций, а также то, как завершать автономную транзакцию и устанавливать точки сохранения;

- обсудим проблемы и ошибки, которых надо остерегаться при использовании автономных транзакций в приложениях.

Пример

Чтобы показать возможности автономных транзакций, я начну с простого примера, демонстрирующего последствия их выполнения. Создадим простую таблицу для сохранения сообщений, а также две процедуры: обычную и оформленную как автономная транзакция. Процедуры будут изменять созданную таблицу. С помощью этих объектов я продемонстрирую, какие изменения остаются (фиксируются) в базе данных в различных ситуациях:

```
tkyte@TKYTE816> create table t (msg varchar2(25));

Table created.

tkyte@TKYTE816> create or replace procedure Autonomous_Insert
  2  as
  3         pragma autonomous_transaction;
  4  begin
  5         insert into t values ('Autonomous Insert');
  6          commit;
  7  end;
  8  /

Procedure created.

tkyte@TKYTE816> create or replace procedure NonAutonomous_Insert
  2  as
  3  begin
  4         insert into t values ('NonAutonomous Insert');
  5          commit;
  6  end;
  7  /

Procedure created.
```

Процедуры просто вставляют свои имена в таблицу сообщений и фиксируют результат. Обратите внимание на использование **PRAGMA AUTONOMOUS_TRANSACTION**. Эта директива указывает серверу, что процедура должна выполняться как новая автономная транзакция, независимо от родительской транзакции. Теперь рассмотрим поведение обычной, не автономной транзакции в анонимном блоке PL/SQL:

```
tkyte@TKYTE816> begin
  2         insert into t values ('Anonymous Block');
  3         NonAutonomous_Insert;
  4          rollback;
  5  end;
  6  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select * from t;
```

```
MSG

Anonymous Block
NonAutonomous Insert
```

Как видите, изменения, выполненные анонимным блоком (вставка строки), были зафиксированы процедурой **NonAutonomous_Insert**. Зафиксированы обе строки данных, и оператору **rollback** оказалось нечего откатывать. Сравните это с поведением хранимой процедуры, оформленной как автономная транзакция:

```
tkyte@TKYTE816> delete from t;

2 rows deleted.

tkyte@TKYTE816> commit;

Commit complete.

tkyte@TKYTE816>begin
  2        insert into t values ('Anonymous Block');
  3         Autonomous_Insert;
  4          rollback;
  5 end;
  6  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select * from t;

MSG

Autonomous Insert
```

В данном случае остаются только изменения, выполненные и зафиксированные в автономной транзакции. Оператор **INSERT**, выполненный в анонимном блоке, откатывается оператором **ROLLBACK** в строке 4. Оператор **COMMIT** в процедуре — автономной транзакции не влияет на родительскую транзакцию, начатую в анонимном блоке. Этот простой пример показывает суть автономных транзакций и их возможности.

При отсутствии автономных транзакций оператор **COMMIT** в процедуре **NonAutonomous_Insert** зафиксировал не только выполненные в ней изменения (результат выполнения оператора **INSERT),**но и все остальные еще не зафиксированные изменения, выполненные в сеансе (например, вставку строки **Anonymous Block,** выполненную в автономном блоке). Оператор отката ничего не отменил, поскольку при вызове процедуры были зафиксированы обе вставки. В случае применения автономных транзакций все меняется. Изменения, выполненные в процедуре, объявленной как **AUTONOMOUS_TRANSACTION,** зафиксированы, однако изменения, выполненные вне автономной транзакции, отменены.

Сервер Oracle для решения своих внутренних задач поддерживает автономные транзакции уже довольно давно. Мы постоянно их видим в форме рекурсивных SQL-операторов. Например, при выборе из последовательности, не находящейся в кэше, автоматически выполняется рекурсивная транзакция, увеличивающая значение последовательности в таблице **SYS.SEQ$.** Изменение значения последовательности немедленно фиксируется и видимо для других транзакций, но обратившаяся к последователь-

ности транзакция при этом не фиксируется. Кроме того, при откате транзакции увеличенное значение последовательности остается — оно не откатывается вместе с транзакцией, поскольку уже зафиксировано. Управление пространством, проверка и другие внутренние действия сервера также выполняются рекурсивно. Просто эта возможность не предлагалась для широкого использования.

Теперь, увидев действие автономной транзакции, давайте разберемся, для чего такие транзакции можно использовать.

Когда использовать автономные транзакции?

В этом разделе мы рассмотрим ряд ситуаций, когда могут пригодиться автономные транзакции.

Проверка, записи которой не могут быть отменены

В прошлом разработчики приложений часто задавали вопрос: "Как зарегистрировать в журнале проверки **попытку** изменить защищенную информацию?". Они хотели не только **предотвратить** попытку изменения, но и сохранить информацию о такой попытке. В прошлом многие (безуспешно) пытались применять для этого триггеры. Триггер срабатывает **при** выполнении изменения; определяет, что пользователь изменяет данные, которые изменять не должен; создает запись в журнале проверки и завершает изменение как неудавшееся (вызывая откат). К сожалению, если изменение не выполняется, откатывается и запись в журнале проверки — неделимость операторов означает "все или ничего". Теперь, с помощью автономных транзакций, можно безопасно сохранить запись о попытке выполнения несанкционированного изменения и откатить само изменение, что позволяет уведомить пользователя: "Вы не имеете права изменять эти данные, и мы зафиксировали вашу попытку их изменить". Интересно отметить, что встроенное средство **AUDIT** сервера Oracle уже давно позволяло регистрировать с помощью автономных транзакций безуспешные попытки изменить информацию. Наличие такой возможности у разработчика позволяет ему создавать собственные, более гибкие системы проверки.

Вот небольшой пример. Скопируем таблицу **EMP** из схемы пользователя **SCOTT** и создадим таблицу для журнала аудита (audit trail table), куда будем записывать **кто** попытался изменить таблицу **EMP** и **когда** была сделана эта попытка, а также **что** именно пытались изменить. Для автоматического сохранения этой информации для таблицы EMP будет создан триггер, оформленный как автономная транзакция:

```
tkyte@TKYTE816> create table emp
  2  as
  3  select * from scott.emp;

Table created.

tkyte@TKYTE816> grant all on emp to scott;
```

```
Grant succeeded.

tkyte@TKYTE816> create table audit_tab
  2  (username   varchar2(30) default user,
  3   timestamp  date default sysdate,
  4   msg        varchar2(4000)
  5  )
  6  /

Table created.
```

Затем необходимо создать триггер для проверки изменений таблицы **EMP**. Обратите внимание на использование автономной транзакции. Этот триггер предотвращает изменение записи о сотруднике любым пользователем, если этот сотрудник не является его подчиненным. Запрос с конструкцией **CONNECT BY** реализует поиск по всей иерархии подчиненных текущего пользователя. Запрос будет проверять, принадлежит ли запись, которую пользователь пытается изменить, одному из его подчиненных:

```
tkyte@TKYTE816> create or replace trigger EMP_AUDIT
  2  before update on emp
  3  for each row
  4  declare
  5      pragma autonomous_transaction;
  6      l_cnt  number;
  7  begin
  8
  9      select count(*)  into l_cnt
 10        from dual
 11       where EXISTS  (select null
 12                        from emp
 13                       where empno =  :new.empno
 14                       start with mgr =  (select empno
 15                                            from emp
 16                                           where ename = USER)
 17                    connect by prior empno - mgr);
 18
 19     if  ( l_cnt = 0 )
 20      then
 21          insert into audit_tab (msg)
 22         values ('Attempt to update ' || :new.empno);
 23          commit;
 24
 25         raise_application_error(-20001,  'Access Denied');
 26      end if;
 27  end;
 28  /

Trigger created.
```

Итак, мы создали таблицу **EMP**, имеющую иерархическую структуру (задаваемую рекурсивным отношением **EMPNO/MGR**). Имеется также таблица **AUDIT_TAB**, в которую будут записываться неудавшиеся попытки изменить информацию. Создан триггер, позволяющий изменять запись о сотруднике только его руководителю или руководителю руководителя (и так далее).

В этом триггере надо обратить внимание на следующее.

- В определении триггера указана прагма **AUTONOMOUS_TRANSACTION.** Весь триггер выполняется как автономная транзакция по отношению к текущей. Вкратце объясню понятие "прагма" (pragma). *Прагма* — это директива компилятору — способ потребовать от компилятора выполнения определенной опции компиляции. Есть и другие прагмы, описание которых можно найти в руководстве *PL/SQL User's Guide and Reference*.
- Триггер по таблице **EMP** читает данные из таблицы **EMP** в запросе. Подробнее о том, почему это существенно, — чуть ниже.
- Триггер фиксирует транзакцию. Раньше это было невозможно — триггеры никогда не фиксировали изменения. На самом деле триггер не фиксирует изменения, вызвавшие его срабатывание. Он фиксирует только изменения, сделанные им самим (добавленную запись проверки).

Давайте теперь рассмотрим, как все это работает:

```
tkyte@TKYTE816> update emp set sal = sal*10;
update emp set sal = sal*10

ERROR at line 1:
OBA-20001: Access Denied
ORA-06512: at "TKYTE.EMP_AUDIT", line 22
ORA-04088: error during execution of trigger 'TKYTE.EMP_AUDIT'

tkyte@TKYTE816> column msg format a30 word_wrapped
tkyte@TKYTE816> select * from audit_tab;

USERNAME                       TIMESTAMP MSG

TKYTE                          15-APR-01 Attempt to update 7369
```

Итак, триггер среагировал на попытку изменения и предотвратил его, записав при этом информацию о попытке в таблицу проверки (обратите внимание, как с помощью значений по умолчанию, заданных после ключевого слова **DEFAULT** в операторе **CREATE TABLE**, в запись автоматически добавлены значения встроенных функций **USER** и **SYSDATE**). Зарегистрируемся от имени пользователя, который имеет право выполнять изменения, и попробуем выполнить ряд операторов:

```
tkyte@TKYTE816>connect scott/tiger

scott@TKYTE816> update tkyte.emp set sal = sal*1.05 where ename = 'ADAMS';

1 row updated.

scott@TKYTE816> update tkyte.emp set sal = sal*1.05 where ename = 'SCOTT';
update tkyte.emp set sal = sal*1.05 where ename = 'SCOTT'
            *
ERROR at line 1:
ORA-20001: Access Denied
ORA-06512: at "TKYTE.EMP_AUDIT", line 22
ORA-04088: error during execution of trigger 'TKYTE.EMP_AUDIT'
```

В стандартной таблице **EMP** сотрудник **ADAMS** подчиняется сотруднику **SCOTT**, поэтому первый оператор **UPDATE** выполняется успешно. Второе изменение (пользователь **SCOTT** пытается поднять самому себе зарплату) не выполняется, поскольку **SCOTT** не является своим руководителем. Если снова зарегистрироваться от имени пользователя, которому принадлежит таблица **AUDIT_TAB**, можно увидеть следующее:

```
scott@TKYTE816> connect tkyte/tkyte
tkyte@TKYTE816> select * from audit_tab;

USER_NAME                      TIMESTAMP MSG

TKYTE                         15-APR-01 Attempt to update 7369
SCOTT                         15-APR-01 Attempt to update 7788
```

Попытка изменения данных пользователем **SCOTT** зарегистрирована в этой таблице. Осталось разобраться, почему так важно, что триггер по таблице **EMP** читает данные **из таблицы EMP?** Это рассматривается в следующем разделе.

Метод, позволяющий избежать ошибки изменяющейся таблицы

Ошибка изменяющейся таблицы (mutating table error) может возникнуть по нескольким причинам. Чаще всего она возникает при попытке читать данные из таблицы, в ответ на изменение которой сработал триггер. В представленном выше примере мы явно читали данные из таблицы, изменение которой вызвало срабатывание триггера. Закомментируем две строки в тексте триггера и попытаемся использовать его следующим образом:

```
tkyte@TKYTE816> create or replace trigger EMP_AUDIT
  2  before update on emp
  3  for each row
  4  declare
  5   --   pragma autonomous_transaction;
  6      l_cnt  number;
  7  begin
  8
  9     select count(*) into l_cnt
 10       from dual
 11      where EXISTS (select null
 12                      from emp
 13                     where empno = :new.empno
 14                    start with mgr = (select empno
 15                                        from emp
 16                                       where ename = USER)
 17                  connect by prior  empno = mgr);
 18
 19     if ( l_cnt = 0 )
 20      then
 21        insert into audit_tab (msg)
 22        values ('Attempt to update ' || :new.empno);
 23          -- commit;
```

```
 24
 25          raise_application_error(-20001,  'Access Denied');
 26     end if;
 27 end;
 28 /

tkyte@TKYTE816> update emp set sal = sal*10;
update emp set sal = sal*10
       *
ERROR at line 1:
ORA-04091: table TKYTE.EMP is mutating, trigger/function may not see it
ORA-06512: at "TKYTE.EMP_AUDIT", line 6
ORA-04088: error during execution of trigger "TKYTE.EMP_AUDIT'
```

Без использования автономных транзакций представленный выше триггер написать сложно, даже если он всего лишь пытается проверить, имеет ли право пользователь изменять данную строку (и не пытается зарегистрировать эту попытку). До появления прагмы **AUTONOMOUS_TRANSACTION** для этого необходимо было создать пакет и три триггера. Это не значит, что во избежание ошибок изменяющейся таблицы во всех случаях следует использовать автономные транзакции — их надо использовать осторожно и четко представлять себе, как обрабатываются транзакции. В разделе "Проблемы" я объясню это подробнее. Ошибка изменяющейся таблицы призвана защитить целостность данных, и очень важно понимать, почему она возникает. Не обманывайте себя: автономные транзакции не устраняют ошибку изменяющейся таблицы в триггерах раз и навсегда!

Выполнение операторов ЯОД в триггерах

Часто задают вопрос: "Как создать объект базы данных при вставке строки в такую-то таблицу?". При этом спрашивают о разных объектах. Иногда хотят создать пользователя базы данных при вставке строки в таблицу, иногда — таблицу или последовательность. Поскольку при выполнении операторов ЯОД непосредственно перед самим оператором и сразу после него выполняется фиксация транзакции (или фиксация и откат, если при выполнении оператора ЯОД произошла ошибка), выполнять эти операторы в триггере было невозможно. Автономные транзакции теперь позволяют это делать.

Раньше приходилось использовать пакет **DBMS_JOB** для выполнения задания с соответствующими операторами ЯОД после фиксации транзакции. Это решение по-прежнему возможно, и **почти всегда оно является корректным и оптимальным.** Использование подпрограмм пакета **DBMS_JOB** для выполнения оператора ЯОД в виде отдельного задания хорошо тем, что позволяет включить операторы ЯОД в транзакцию. Если триггер поставил задание на выполнение, и это задание создало учетную запись пользователя, при откате родительской транзакции поставленное в очередь задание по созданию учетной записи пользователя тоже будет отменено. Строка, представляющая задание, будет "удалена". Не останется ни записи в таблице, ни учетной записи в системе. Если в этом случае использовать автономные транзакции, учетная запись в базе данных будет создана, а записи в таблице не будет. Недостатком подхода на базе пакета **DBMS_JOB** является неизбежное небольшое отставание между моментом фиксации транзакции и запуском задания. Учетная запись пользователя будет создана вскоре пос-

ле фиксации, но не сразу. В зависимости от требований, можно использовать тот или иной метод. Повторю еще раз, что почти в любом случае можно найти аргументы в пользу пакета **DBMS_JOB.**

В качестве примера выполнения операторов ЯОД в триггере рассмотрим ситуацию, когда необходимо создавать учетную запись пользователя базы данных при вставке строки в таблицу и удалять эту запись при удалении соответствующей строки. В представленном ниже примере я буду использовать свой способ, позволяющий избежать ситуаций, когда учетная запись пользователя создана, а строки в таблице нет, или строка в таблице осталась, а учетная запись пользователя удалена. Этот способ основан на использовании триггера **INSTEAD OF** для представления таблицы **APPLICATION_USERS_TBL.** Триггеры **INSTEAD OF** — удобное средство, позволяющее задать действия, выполняемые при изменении строк представления **вместо** стандартных действий сервера Oracle. В главе 20, посвященной использованию объектно-реляционных средств, будет продемонстрировано, как с помощью триггеров **INSTEAD OF** обеспечить изменение сложных представлений, которые сервер Oracle обычно изменять не позволяет. Мы будем использовать эти триггеры для создания учетной записи пользователя и вставке строки в реальную таблицу (или удаления учетной записи пользователя и соответствующей строки таблицы). Этот метод гарантирует, что либо создана учетная запись и строка вставлена, либо ни то, ни другое не выполнено. Если бы триггер был создан только по самой таблице, мы не могли бы этого гарантировать, а вот **представление** поможет нам связать эти два события. Вместо вставки и удаления данных из реальной физической таблицы все приложения будут вставлять и удалять данные из представления. Для представления будет создан триггер **INSTEAD OF,** так что все изменения можно будет выполнить последовательно, в процедурном коде. Это позволит гарантировать, что если строка существует в реальной таблице, то и учетная запись пользователя тоже создана. Если строка удалена из реальной таблицы, то удалена и учетная запись. Как этого добиться, лучше всего продемонстрирует пример. Я буду объяснять существенные детали, как только мы до них доберемся.

Начнем с создания схемы, в которой будут храниться объекты приложения:

```
tkyte@TKYTE816> create user demo_ddl identified by demo_ddl;

User created.

tkyte8TKYTE816> grant connect, resource to demo_ddl with admin option;

Grant succeeded.

tkyte@TKYTE816> grant create user to demo_ddl;
Grant succeeded.

tkyte@TKYTE816> grant drop user to demo_ddl;
Grant succeeded.

tkyte@TKYTE816> connect demo_ddl/demo_ddl
demo_ddl@TKYTE816>
```

Итак, мы только что создали учетную запись пользователя и хотим, чтобы этот пользователь мог предоставлять привилегии **CONNECT** и **RESOURCE** другим пользователям. (Привилегии **CONNECT** и **RESOURCE** использованы для простоты. Используйте те привилегии, которые необходимо.) Для того чтобы предоставлять эти привилегии дру-

гим, сам он должен иметь привилегии **CONNECT** и **RESOURCE** с опцией **WITH ADMIN OPTION**. Кроме того, поскольку предполагается создание и удаление учетных записей в триггере, надо предоставить соответствующие привилегии **CREATE** и **DROP** непосредственно, как показано выше. Эти привилегии должны быть предоставлены пользователю непосредственно, а не через роль, поскольку триггеры всегда выполняются с правами создателя, а в этом режиме роли не учитываются (подробнее об этом см. в главе 23).

Теперь создадим таблицу приложения, в которой будет храниться информация о пользователях. Для этой таблицы мы создадим триггер для событий **BEFORE INSERT** или **DELETE**. Этот триггер будет гарантировать, что ни один пользователь (включая владельца) не сможет вставить или удалить данные из этой таблицы непосредственно. Нам надо, чтобы все вставки/удаления выполнялись через представление и чтобы при этом обязательно выполнялись операторы ЯОД.

В представленном ниже коде **MY_CALLER** — небольшая функция, которую я часто использую (совместно с подпрограммой **WHO_CALLED_ME**). Код этих подпрограмм можно найти в приложении *"Основные стандартные пакеты"* в конце книги, в разделе, посвященном пакету **DBMS_UTILITY**. Эта функция просто возвращает имена процедур/функций/триггеров, вызвавших ее. Если **MY_CALLER** вызвана не из триггера по представлениям (который еще надо создать), выполнение этой операции запрещено.

```
demo_ddl@TKYTE816> create table application_users_tbl
  2  (uname       varchar2(30) primary key,
  3   pw           varchar2(30),
  4  role_to_grant varchar2(4000)
  5  );

Table created.

demo_ddl@TKYTE816> create or replace trigger application_users_tbl_bid
  2 before insert or delete on application_users_tbl
  3 begin
  4    if (my_caller not in ('DEMO_DDL.APPLICATION_USERS_IOI',
  5                          'DEMO_DDL.APPLICATION_USERS_IOD'))
  6     then
  7     raise_application_error(-20001, 'Cannot insert/delete directly');
  8     end if;
  9 end;
 10 /

Trigger created.
```

Создадим представление с триггерами **INSTEAD OF**, которые и будут выполнять необходимые действия. А для представления создадим триггер **INSTEAD OF INSERT**, позволяющий создавать учетные записи. Создадим также для представления триггер **INSTEAD OF DELETE**. Он будет вызывать выполнение оператора **DROP USER**. Можно расширить пример, воспользовавшись триггерами **INSTEAD OF UPDATE**, которые позволяют добавлять роли и изменять пароли с помощью простых операторов UPDATE.

Срабатывая, триггер **INSTEAD OF INSERT** выполняет два оператора:

- оператор, аналогичный **GRANT CONNECT, RESOURCE TO SOME_USERNAME IDENTIFIED BY SOME_PASSWORD**;
- оператор вставки **INSERT** в созданную ранее таблицу **APPLICATION_USERS_TBL**.

Причина использования оператора **GRANT** вместо последовательности **CREATE USER**, а затем уже **GRANT** в том, что при этом операторы **COMMIT**, **CREATE USER**, **GRANT** и **COMMIT** выполняются за один шаг. Причем, если этот единственный оператор (предоставления привилегий) завершится неудачно, не придется удалять учетную запись пользователя вручную. Оператор **CREATE USER** может выполниться успешно, а при выполнении **GRANT** может произойти сбой. Ошибки, возникающие при выполнении оператора **GRANT**, все равно надо перехватывать, чтобы удалить только что вставленную строку.

Поскольку операторы **INSERT** и **GRANT** выполняются для каждой вставляемой в представление строки, можно с уверенностью утверждать: если строка существует в реальной таблице, значит, соответствующая учетная запись успешно создана, а если нет строки, то нет и учетной записи. Потенциальная возможность сбоя все же остается, полностью избавиться от нее нельзя. Если после вставки строки в таблицу **APPLICATION_USERS_TBL** оператор **GRANT** не срабатывает, а удалить только что вставленную строку невозможно (из-за сбоя системы или недоступности табличного пространства, содержащего таблицу **APPLICATION_USERS_TBL** и т.п.), мы получим рассогласование. Не забывайте, что оператор GRANT на самом деле представляет собой тройку операторов **COMMIT/GRANT/COMMIT**, как и все операторы ЯОД, поэтому перед сбоем оператора **GRANT** результат оператора **INSERT** уже зафиксирован. Временной промежуток, когда это может случиться, однако, очень мал, чтобы можно было без опасений пользоваться этим методом.

Теперь создадим представление и описанные выше триггеры:

```
demo_ddl@TKYTE816> create or replace view
  2 application_users
  3  as
  4 select * from application_users_tbl
  5  /

View created.

demo_ddl@TKYTE816> create or replace trigger application_users_IOI
  2 instead of insert on application_users
  3 declare
  4  pragma autonomous_transaction;
  5 begin
  6  insert into application_users_tbl
  7  (uname, pw, role_to_grant)
  8   values
  9  (upper(:new.uname),   :new.pw,  :new.role_to_grant);
 10
 11  begin
 12    execute immediate
 13     'grant ' || :new.role_to_grant ||
 14     ' to ' || :new.uname ||
```

```
 15    '    identified by ' || :new.pw;
 16   exception
 17     when others then
 18         delete from application_users_tbl
 19           where uname = upper(:new.uname);
 20          commit;
 21          raise;
 22   end;
 23 end;
 24 /

Trigger created.
```

Итак, триггер **INSTEAD OF INSERT** по этой таблице сначала вставляет строку в таблицу **APPLICATION_USERS_TBL**. Затем он выполняет оператор **GRANT** для создания учетной записи пользователя. Оператор **GRANT** фактически представляет собой тройку **COMMIT/GRANT/COMMIT**, так что после его выполнения строка в таблице **APPLICATION_USER_TBL** зафиксирована. Если оператор **GRANT** успешно выполнен, значит, автономная транзакция зафиксирована, и работа триггера завершается. Если же оператор **GRANT** не сработал (потому что учетная запись пользователя уже существует, имя пользователя — недопустимое и т.д.), мы перехватываем ошибку, явно удаляем вставленную строку и фиксируем удаление. Затем мы снова возбуждаем исключительную ситуацию.

В данном случае мы выполняем оператор **INSERT**, а затем — оператор ЯОД, поскольку отменить **INSERT** намного проще, чем отменить создание учетной записи пользователя (для отмены сделанного предпочтительнее выполнять оператор **DELETE**, а не DROP). В конечном итоге триггер гарантирует, что либо вставлена строка в таблицу **APPLICATION_USERS_TBL** и создана соответствующая учетная запись пользователя, либо ни одно из этих действий не выполнено.

Теперь перейдем к триггеру **INSTEAD OF DELETE**, удаляющему строку и учетную запись пользователя:

```
demo_ddl@TKYTE816> create or replace trigger application_users_IOD
  2 instead of delete on application_users
  3 declare
  4  pragma autonomous_transaction;
  5 begin
  6     execute immediate 'drop user ' || :old.uname;
  7     delete from application_users_tbl
  8      where uname = :old.uname;
  9      commit;
 10 end;
 11 /

Trigger created.
```

Я умышленно изменил порядок выполнения действий. В этом триггере сначала выполняется оператор ЯОД, а потом — оператор ЯМД, а раньше было наоборот. Причина снова связана с простотой восстановления в случае ошибки. Если оператор **DROP USER** не срабатывает, отменять ничего не нужно. Вероятность сбоя при выполнении оператора **DELETE** нулевая. Нет никаких требований целостности, которые могли бы поме-

шать удалению строки. При большой вероятности неудачного выполнения оператора **DELETE** из-за имеющихся требований целостности ссылок, порядок выполнения действий можно изменить (по аналогии с триггером **INSTEAD OF INSERT**).

Теперь протестируем решение: вставим запись о пользователе в представление, проверим, создана ли учетная запись пользователя, и, наконец, удалим учетную запись пользователя.

```
demo_ddl@TKYTE816> select * from all_users where username = 'NEW_USER';

no rows selected

demo_ddl@TKYTE816> insert into application_users values
  2  ('new_user', 'pw', 'connect, resource');

1 row created.

demo_ddl@TKYTE816> select * from all_users where username = 'NEW_USER';

USERNAME                          USER_ID CREATED

NEW_USER                              235 15-APR-01

demo_ddl@TKYTE816> delete from application_users where uname = 'NEW_USER';

1 row deleted.

demo_ddl@TKYTE816> select * from all_users where username = 'NEW_USER';

no rows selected
```

(Полученное вами при выполнении этого примера значение **USER_ID** скорее всего будет отличаться от 235. Не удивляйтесь — это вполне объяснимо.) Наконец, убедимся, что нельзя удалять и вставлять данные непосредственно в "реальную" таблицу.

```
demo_ddl@TKYTE816> insert into application_users_tbl values
  2  ('new_user', 'pw', 'connect, resource');
insert into application_users_tbl values
            *

ERROR at line 1:
ORA-20001: Cannot insert/delete directly
ORA-06512: at "DEM0_DDL.APPLICATION_USERS_TBL_BID", line 5
ORA-04088: error during execution of trigger
->'DEMO_DDL.APPLICATION_USERS_TBL_BID'


demo_ddl@TKYTE816> delete from application_users_tbl;
delete from application_users_tbl
            *
ERROR at line 1:
ORA-20001: Cannot insert/delete directly
ORA-06512: at "DEMO_DDL.APPLICATION_USERS_TBL_BID", line 5
ORA-04088: error during execution of trigger
->'DEMO_DDL.APPLICATION_USERS_TBL_BID'
```

Вот и все. Триггеры обеспечивают добавление и удаление учетных записей при вставке и удалении строк из таблицы базы данных. С помощью триггеров **INSTEAD OF** можно обеспечить безопасность данной операции за счет выполнения компенсирующих транзакций, гарантируя при необходимости синхронность изменения таблиц приложения и

выполнения операторов ЯОД. Можно пойти еще дальше и создать триггеры на события базы данных, срабатывающие при удалении учетной записи пользователя с помощью оператора **DROP,** чтобы исключить удаление учетной записи без изменения представления.

Запись в базу данных

В сервере Oracle 7.1 впервые появилась возможность расширять набор встроенных функций SQL с помощью функций, реализованных на языке PL/SQL. Это очень мощная возможность, особенно теперь, когда эти функции можно писать не только на языке PL/SQL, но и на Java или C. В прошлом функции, вызываемые в операторах SQL, не должны были изменять состояние базы данных (Write No Database State — WNDS). Если функция выполняла операторы **INSERT, UPDATE, DELETE, CREATE, ALTER, COMMIT** и т.д. или вызывала процедуру или функцию, выполняющую подобные действия, ее нельзя было использовать в SQL-операторах.

С помощью автономных транзакций мы теперь можем изменять состояние базы данных в функциях, вызываемых в SQL-операторах. Это требуется не так уж часто:

- **строгая проверка;** необходимо знать, какие данные видел каждый из пользователей, или надо записать идентификатор каждой записи, запрошенной у системы;
- **средство создания отчетов позволяет выполнять только SQL-операторы** SELECT; абсолютно необходимо по ходу построения отчета вызывать хранимую процедуру, выполняющую ряд вставок (например, заполняющую таблицу параметров для другого отчета).

Давайте рассмотрим, как решить эти проблемы.

Строгая проверка

Я знаю ряд правительственных учреждений, где из соображений конфиденциальности необходимо регистрировать, кто видел различные части записи. Например, налоговая служба накапливает детальные данные о том, сколько вы заработали, что вам принадлежит и т.п. Когда кто-то запрашивает данные для того или иного лица и видит эту конфиденциальную информацию, необходимо зарегистрировать это действие в журнале проверки. По этому журналу со временем можно будет понять, не получают ли сотрудники записи, которые не имеют права получать, или ретроспективно определить, кто обращался к соответствующим записям в случае публикаций в прессе или других утечек информации.

С помощью автономных транзакций и представлений можно реализовать такую проверку ненавязчиво и абсолютно прозрачно для пользователей, независимо от используемых ими инструментальных средств. Они не смогут обойти эту систему проверки, и при этом она не будет им мешать. При этом, естественно, для выполнения запросов понадобятся дополнительные ресурсы, но это вполне подходит для ситуаций, когда записи выбираются по одной, а не сотнями или тысячами. С учетом этих ограничений реализация получается достаточно простой. Используя таблицу **EMP** в качестве шаблона, можно реализовать проверку по столбцам **HIREDATE, SALARY** и **COMMISSION,** и когда кто-либо просматривает, например, данные о зарплате **(SALARY),** мы будем знать, кто их просматривал и **какие** именно записи увидел. Начнем с создания таблицы для

журнала проверки обращений к таблице EMP, которую мы скопировали из схемы пользователя SCOTT ранее в этой главе:

```
tkyte@TKYTE816> create table audit_trail
  2  (username  varchar2(30),
  3   pk        number,
  4   attribute varchar2(30),
  5   dataum    varchar2(255),
  6   timestamp date
  7  )
  8  /

Table created.
```

Затем создадим ряд перегруженных функций в пакете, реализующем проверку. Каждая из этих функций принимает в качестве аргумента значение первичного ключа выбираемой строки, а также значение и имя столбца. Перегруженные функции используются, чтобы даты сохранялись как даты, а числа — как числа, что позволяет преобразовать их в стандартный формат (в строку) для хранения в созданном выше столбце **DATAUM**:

```
tkyte@TKYTE816> create or replace package audit_trail_pkg
  2  as
  3      function record(p_pk in number,
  4                      p_attr in varchar2,
  5                      p_dataum in number)  return number;
  6      function record(p_pk in number,
  7                      p_attr in varchar2,
  8                      p_dataum in varchar2)  return varchar2;
  9      function record(p_pk in number,
 10                      p_attr in varchar2,
 11                      p_dataum in date)  return date;
 12  end;
 13  /

Package created.
```

Итак, теперь все готово для реализации тела пакета. Каждая из объявленных выше функций **RECORD** вызывает внутреннюю процедуру **LOG**. Процедура **LOG** выполняется как автономная транзакция, вставляющая и фиксирующая запись в таблицу проверки. Обратите внимание, в частности, на то, как представленная ниже функция **RECORD**, возвращающая данные типа **DATE** автоматически преобразует дату в строку с сохранением времени:

```
tkyte@TKYTE816> create or replace package body audit_trail_pkg
  2  as
  3
  4  procedure log(p_pk in number,
  5                p_attr in varchar2,
  6                p_dataum in varchar2)
  7  as
  8      pragma autonomous_transaction;
  9  begin
```

```
 10      insert into audit_trail values
 11     (user, p_pk, p_attr, p_dataum, sysdate);
 12      commit;
 13  end;
 14
 15  function record(p_pk in number,
 16                  p_attr in varchar2,
 17                  p_dataum in number)  return number
 18  is
 19  begin
 20     log(p_pk, p_attr, p_dataum);
 21      return p_dataum;
 22  end;
 23
 24  function record(p_pk in number,
 25                  p_attr in varchar2,
 26                  p_dataum in varchar2)  return varchar2
 27  is
 28  begin
 29     log(p_pk, p_attr, p_dataum);
 30      return p_dataum;
 31  end;
 32
 33  function record(p_pk in number,
 34                  p_attr in varchar2,
 35                  p_dataum in date)  return date
 36  is
 37  begin
 38      log(p_pk, p_attr,
 39         to_char (p_dataum, 'dd-mon-yyyy hh24 :mi:ss')) ;
 40      return p_dataum;
 41  end;
 42
 43  end;
 44  /

Package body created.

tkyte@TKYTE816> create or replace view emp_v
  2  as
  3  select empno , ename, job,mgr,
  4         audit_trail_pkg.record(empno, 'sal', sal) sal,
  5         audit_trail_pkg.record(empno, 'comm', comm) comm,
  6         audit_trail_pkg.record(empno, 'hiredate', hiredate) hiredate,
  7         deptno
  8    from emp
  9  /

View created.
```

Мы создали представление, возвращающее три столбца — **HIREDATE**, **SAL** и **COMM** - через PL/SQL-функцию. Эта PL/SQL-функция записывает, кто, что и когда просматривал. Это представление подходит для непосредственных поисковых запросов вида:

```
tkyte@TKYTE816> select empno, ename, hiredate, sal, comm, job
  2    from emp_v where ename = 'KING';

     EMPNO ENAME      HIREDATE          SAL       COMM JOB

      7839 KING       17-NOV-81        5000            PRESIDENT

tkyte@TKYTE816> column username format a8
tkyte@TKYTE816> column pk format 9999
tkyte@TKYTE816> column attribute format a8
tkyte@TKYTE816> column dataum format a20

tkyte@TKYTE816> select * from audit_trail;

USERNAME    PK ATTRIBUT DATAUM                TIMESTAMP

TKYTE     7839 hiredate 17-nov-1981 00:00:00 15-APR-01
TKYTE     7839 sal      5000                 15-APR-01
TKYTE     7839 comm                          15-APR-01

tkyte@TKYTE816> select empno, ename from emp_v where ename = 'BLAKE'

     EMPNO ENAME

      7698 BLAKE

tkyte@TKYTE816> select * from audit_trail;

USERNAME    PK ATTRIBUT DATAUM                TIMESTAMP

TKYTE     7839 hiredate 17-nov-1981 00:00:00 15-APR-01
TKYTE     7839 sal      5000                 15-APR-01
TKYTE     7839 comm                          15-APR-01
```

Как видно по этим результатам, пользователь **TKYTE** просматривал столбцы **HIREDATE**, **SAL** и **COMM** в указанный день. По второму запросу информация из этих столбцов не получена, поэтому дополнительные записи в журнал проверки не внесены.

Подобное представление, как уже было сказано, подходит для простых справочных запросов, потому что в некоторых случаях оно регистрирует "лишние" попытки доступа. Бывают случаи, когда данные представления показывают, что кто-то просматривал фрагмент информации, тогда как фактически он его не видел. Он был отброшен в дальнейшем в сложном запросе или агрегирован в определенное значение, не имеющее отношения к данному лицу. Следующий пример показывает, что при агрегировании или использовании столбца в конструкции **WHERE** в журнал проверки вносится запись о том, что столбец просмотрен.

Начнем с очистки таблицы журнала проверки, чтобы происходящее стало очевидным:

```
tkyte@TKYTE816> delete from audit_trail;

3 rows deleted.

tkyte@TKYTE816> commit;

Commit complete.

tkyte@TKYTE816> select avg(sal) from emp_v;
```

```
  AVG(SAL)

2077.14286
tkyte@TKYTE816> select * from audit_trail;
USERNAME    PK ATTRIBUT DATAUM                TIMESTAMP

TKYTE     7499 sal      1600                  15-APR-01

TKYTE     7934 sal      1300                  15-APR-01
14 rows selected.
tkyte@TKYTE816> select ename from emp_v where sal >= 5000;
ENAME

KING
tkyte@TKYTE816> select * from audit_trail;
USERNAME    PK ATTRIBUT DATAUM                TIMESTAMP

TKYTE     7499 sal      1600                  15-APR-01

TKYTE     7934 sal      1300                  15-APR-01
28 rows selected.
```

При выполнении запроса с агрегированием зарегистрирован просмотр каждого значения зарплаты, которое было просмотрено для получения среднего, **AVG(SAL)**. Запрос **WHERE SAL >= 5000** записал каждую зарплату, которая просматривалась для получения ответа. Для таких случаев хороших решений нет, разве что не использовать представления в такого рода запросах. Для получения значения **AVG(SAL)** можно использовать представление, включающее столбец **SAL** и, возможно, другие данные. Запрос должен выполняться к представлению, не связывающему значение в столбце **SAL** с определенным лицом. Оно позволит просматривать зарплаты, но не узнавать, кто их получает. Проверить условие **SAL >= 5000**, не записав каждую зарплату, сложно. Я бы использовал хранимую процедуру, возвращающую курсорную переменную, **REF CURSOR**. В этой хранимой процедуре можно было бы обращаться к таблице **EMP** и проверять условия по столбцу **SAL**, но выбирать любую информацию, за исключением столбца **SAL**. Пользователь не узнает, сколько именно получил тот или иной человек, он узнает только, что зарплата превышала определенное значение. Представлением **EMP_V** можно будет пользоваться только при необходимости получить одновременно и личную информацию (значения столбцов **EMPNO** и **ENAME**), и зарплату (**SAL**).

Выполнение операторов ЯМД в SQL в данном случае специфично, и делать это надо осторожно.

Когда среда позволяет выполнять только операторы SELECT

Это действительно удобное использование автономных транзакций в SQL-операторах. Во многих случаях используются инструментальные средства, позволяющие выпол-

нять только операторы SELECT или даже простые операторы типа **INSERT,** но на самом деле необходимо вызывать хранимые процедуры, а этого подобные средства не позволяют. Автономные же транзакции позволяют вызывать любую хранимую процедуру или функцию с помощью SQL-оператора **SELECT.**

Предположим, создана хранимая процедура, вставляющая ряд значений в таблицу для того, чтобы на их основе ограничить набор строк, выдаваемых последующим запросом в том же сеансе. Если значений в этой таблице нет, отчет создать нельзя. Работать приходится в среде, не позволяющей выполнять хранимые процедуры, а только обычные SQL-операторы. Как же поступить, если выполнить эту процедуру **необходимо.** Следующий пример демонстрирует возможное решение:

```
tkyte@TKYTE816> create table report_parm_table
  2  (session_id   number,
  3       arg1       number,
  4   arg2           date
  5  )
  6  /

Table created.

tkyte@TKYTE816> create or replace
  2  procedure set_up_report(p_arg1 in number, p_arg2 in date)
  3  as
  4  begin
  5     delete from report_parm_table
  6    where session_id = sys_context('userenv','sessionid');
  7
  8     insert into report_parm_table
  9      (session_id, arg1, arg2)
 10     values
 11   (sys_context('userenv','sessionid'),   p_arg1,   p_arg2);
 12  end;
 13  /

Procedure created.
```

Итак, имеется хранимая процедура, изменяющая состояние базы данных; она входит в уже существующую систему. Мы хотим вызывать ее из SQL-оператора **SELECT**, поскольку это единственно доступный способ. Процедуру **SET_UP_REPORT** необходимо "обернуть" в небольшую PL/SQL-функцию, поскольку в SQL-операторах можно вызывать только функции. Кроме того, такая "обертка" нужна, чтобы можно было задать прагму **AUTONOMOUS_TRANSACTION**:

```
tkyte@TKYTE816> create or replace
  2  function set_up_report_F(p_arg1 in number, p_arg2 in date)
  3  return number
  4  as
  5     pragma autonomous_transaction;
  6  begin
  7    set_up_report(p_arg1,    p_arg2);
  8       commit;
```

```
  9      return 1;
 10  exception
 11      when others then
 12          rollback;
 13          return 0;
 14  end;
 15  /

Function created.

tkyte@TKYTE816> select set_up_report_F(1, sysdate) from dual
  2  /

SET_UP_REPORT_F(1,SYSDATE)

                          1

tkyte@TKYTE816> select * from report_parm_table
  2
```

Интересно посмотреть, что произойдет, если попытаться вызвать эту функцию в SQL-операторе, не объявив ее как автономную транзакцию. Если перекомпилировать представленную выше функцию без прагмы, при выполнении оператора будут получены следующие сообщения:

```
tkyte@TKYTE816> select set_up_report_F(1, sysdate) from dual
  2  /
select set_up_report_F(1, sysdate) from dual
       *
ERROR at line 1:
ORA-14552: cannot perform a DDL, commit or rollback inside a query or DML
ORA-06512: at "TKYTE.SET_UP_REPORT_F", line 10
ORA-14551: cannot perform a DML operation inside a query
ORA-06512: at line 1
```

Именно этого и позволяет избежать автономная транзакция. Итак, мы создали функцию, которую можно вызывать из SQL-операторов и которая вставляет строку в базу данных. Важно помнить, что эта функция должна обязательно зафиксировать (или откатить) транзакцию до завершения работы — в противном случае будет получено сообщение об ошибке **ORA-06519** (подробнее см. далее в разделе *"Возможные сообщения об ошибках")*. Кроме того, функция обязательно **должна** возвращать значение. Моя функция возвращает 1 в случае успешного выполнения и 0 — в случае неудачи. Кроме того, надо учитывать, что функция может иметь только параметры, передаваемые в режиме IN, — никаких параметров **IN/OUT** или **OUT.** Дело в том, что SQL не позволяет задавать параметры с этими режимами передачи.

Я хочу рассказать о проблемах при использовании описанного подхода. Обычно я описываю проблемы в конце главы, но в данном случае проблемы непосредственно связаны с изменением базы данных операторами **SELECT.** У таких изменений могут быть опасные побочные эффекты, связанные со способом оптимизации и выполнения запросов. Представленный выше пример был достаточно безопасен. Таблица **DUAL** — однострочная, мы выбирали значение функции, и вызываться функция будет только один раз. Не было никаких соединений, предикатов, сортировок и побочных эффектов. Она

должна работать надежно. Иногда функция вызывается меньшее или большее количество раз, чем предполагалось. Чтобы продемонстрировать это, я прибегну к несколько надуманному примеру. Используем простую таблицу **COUNTER**, которую автономная транзакция будет обновлять при каждом выполнении. Таким образом, мы сможем выполнять запросы и видеть, сколько раз вызывалась функция:

```
tkyte@TKYTE816> create table counter (x int);

Table created.

tkyte@TKYTE816> insert into counter values (0);

1 row created.

tkyte@TKYTE816> create or replace function f return number
  2  as
  3          pragma autonomous_transaction;
  4  begin
  5          update counter set x = x+1;
  6          commit;
  7          return 1;
  8  end;
  9  /

Function created.
```

Итак, мы создали таблицу **COUNTER** и функцию. При каждом вызове функции F значение X будет увеличиваться на 1. Давайте попробуем:

```
tkyte@TKYTE816> select count(*)
  2    from (select * from emp)
  3  /

  COUNT(*)

        14

tkyte@TKYTE816> select * from counter;

         X

         0
```

Как видите, функция ни разу не вызывалась, хотя должна была вызываться 14 раз. Чтобы продемонстрировать, что функция F работает, выполним следующий оператор:

```
tkyte@TKYTE816> select count(*)
  2  from (select f from emp union select f from emp)
  3  /

  COUNT(*)

         1

tkyte@TKYTE816> select * from counter;

         X

        28
```

Именно этого мы и ожидали. Функция F была вызвана 28 раз (14 из них — для запросов в операторе **UNION**). Поскольку при выполнении оператора **UNION** в качестве побочного эффекта происходит сортировка для поиска неповторяющихся значений (**SORT DISTINCT**), функция **COUNT(*)** по объединению дает 1 (и это правильно), а функция, как и ожидалось, вызывалась 28 раз. Слегка изменим запрос:

```
tkyte@TKYTE816>update counter set x = 0;

1 row updated.

tkyte@TKYTE816> commit;

Commit complete.

tkyte@TKYTE816> select f from emp union ALL select f from emp
  2  /

         F

         1

         1

28 rows selected.

tkyte@TKYTE816> select * from counter;

         X

        32
```

Запрос вернул 28 строк, но наша функция была вызвана 32 раза! Помните о побочных эффектах. Сервер Oracle не гарантирует, что функция вообще будет вызвана (вспомните первый пример) или будет вызвана определенное количество раз (последний пример). Будьте особенно осторожны при использовании таких изменяющих базу данных функций в SQL-операторах, если используется другая таблица, кроме **DUAL**, выполняются соединения, сортировки и т.п. Результаты могут оказаться неожиданными.

Разработка модульного кода

Автономные транзакции также позволяют повысить модульность кода. Традиционно создается набор пакетов, выполняющих некоторые действия. Эти действия представляют собой ряд изменений в базе данных, которые, в зависимости от результатов, фиксируются или откатываются. Все это замечательно, если надо заботиться только о своих процедурах. В крупномасштабном приложении, однако, ваш простой пакет будет далеко не единственным компонентом. Скорее всего, он будет лишь небольшой частью общей картины.

В типичной процедуре фиксируются не только выполненные в ней изменения, но и все не зафиксированные изменения, выполненные в сеансе до вызова этой процедуры. При фиксации постоянными делаются все эти изменения. Проблема в том, что выполнявшиеся до вызова процедуры изменения могут быть не завершены. Поэтому выполнение оператора COMMIT может привести к ошибке в вызывающей процедуре. Она

уже не сможет откатить выполненные изменения в случае сбоя, даже "не подозревая" об этом.

С помощью автономных транзакций теперь можно создавать самодостаточные подпрограммы, выполняющие транзакции, которые не влияют на состояние вызывающих транзакций. Это может быть весьма полезно для многих типов подпрограмм, в частности, реализующих проверку, журнализацию и другие служебные функции. Так можно создавать код, который безопасно вызывать из различных сред, не влияя деструктивно на функционирование этих сред. Использование автономных транзакций для проверки, журнализации и других служебных функций — вполне оправдано. Лично я считаю фиксацию транзакций в хранимых процедурах обоснованной только в перечисленных случаях. Я считаю, что фиксировать транзакции должно только клиентское приложение. Необходимо быть осторожным и использовать автономные транзакции правильно. Если код должен вызываться как логическая часть более масштабной транзакции (например, если создан пакет **ADDRESS_UPDATE** для системы учета кадров), то оформлять его в виде автономной транзакции нельзя. Во внешней среде необходимо обеспечить возможность вызова этого пакета и других соответствующих пакетов, а затем зафиксировать (или отменить) все изменения в целом. Поэтому при использовании автономной транзакции контроль из вызывающей среды невозможен, как и построение больших транзакций из меньших составных частей. Кроме того, автономная транзакция не видит незафиксированных изменений, выполненных в вызывающей транзакции. Подробнее это описано в разделе *"Как работают автономные транзакции"*. Это означает, что для автономной транзакции незафиксированные изменения остальной кадровой информации будут невидимы. Для корректного использования автономных транзакций необходимо четко представлять все возможные варианты использования создаваемого кода.

Как работают автономные транзакции

В этом разделе я опишу, как работают автономные транзакции и чего от них можно ожидать. Мы изучим последовательность действий при выполнении автономных транзакций. Мы также рассмотрим, как автономные транзакции влияют на область действия различных элементов — переменных пакетов, параметров сеансов, параметров базы данных и блокировок. Мы поговорим о том, как правильно завершать автономную транзакцию, и о точках сохранения.

Выполнение транзакции

Выполнение автономной транзакции начинается с ключевого слова **BEGIN** и завершается ключевым словом **END**. То есть, при наличии следующего блока кода:

```
declare
    pragma autonomous_transaction;
    X number default func;              (1)
begin                                   (2)

end;                                    (3)
```

автономная транзакция начинается со строки (2), а не (1). Она начинается с первого выполняемого оператора. Если **FUNC** — функция, выполняющая изменения в базе данных, — эти изменения не являются частью автономной транзакции. Они — часть родительской транзакции. Кроме того, порядок следования элементов в разделе **DECLARE** блока не имеет значения: конструкция **PRAGMA** может быть как первой, так и последней. Весь раздел **DECLARE** блока является частью родительской транзакции, а не автономной. Следующий пример поможет это прояснить:

```
tkyte@TKYTE816> create table t (msg varchar2(50));

Table created.

tkyte@TKYTE816> create or replace function func return number
  2  as
  3  begin
  4          insert into t values
  5          ('Строка вставлена функцией FUNC);
  6          return 0;
  7  end;
  8  /

Function created.
```

Итак, имеется функция, изменяющая базу данных. Давайте теперь вызовем эту функцию в разделе **DECLARE** блока, оформленного как автономная транзакция:

```
tkyte@TKYTE816> declare
  2          x  number default func;
  3          pragma autonomous_transaction;
  4  begin
  5          insert into t values
  6          ('Строка вставлена анонимный блоком');
  7          commit;
  8  end;
  9  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select * from t;

MSG

Строка вставлена функцией FUNC
Строка вставлена анонимным блоком
```

Пока что обе строки есть. Однако одна из строк еще не зафиксирована. В этом можно убедиться, выполнив откат:

```
tkyte@TKYTE816> rollback;

Rollback complete.

tkyte@TKYTE816> select * from t;

MSG

Строка вставлена анонимным блоком
```

Как видите, после выполнения анонимного блока кажется, что обе строки вставлены в таблицу T и зафиксированы. Это, однако, ошибочное представление. Строка, вставленная функцией, на самом деле еще не зафиксирована. Она — часть родительской, еще не завершенной транзакции. Выполняя откат, мы убеждаемся, что она исчезает, а вот строка, вставленная в автономной транзакции, остается.

Итак, автономная транзакция начинается с **первого** за конструкцией **PRAGMA** ключевого слова **BEGIN** и действует в пределах соответствующего блока. Любые функции или процедуры, которые вызываются в автономной транзакции, триггеры, срабатывание которых она вызывает, и т.д. являются частью этой автономной транзакции, и выполненные ими изменения будут зафиксированы или отменены вместе с ней.

Автономные транзакции могут быть вложенными — в автономной транзакции можно начать новую автономную транзакцию. Вложенные автономные транзакции обрабатываются точно так же, как родительская, — они начинаются с первого ключевого слова **BEGIN,** действуют вплоть до соответствующего ключевого слова **END** и полностью независимы от родительской транзакции. Единственное ограничение на глубину вложенности автономных транзакций задается параметром инициализации **TRANSACTIONS**, который определяет, сколько одновременных транзакций может поддерживать сервер. Обычно это значение равно количеству сеансов (**SESSIONS**), умноженному на 1,1; если планируется интенсивно использовать автономные транзакции, значение этого параметра должно быть увеличено.

Область действия

Под *областью действия* подразумевается возможность получать значения различных элементов базы данных. В данном случае нас интересуют четыре элемента. Рассмотрим поочередно области действия:

- переменных пакетов;
- установок/параметров сеанса;
- изменений в базе данных;
- блокировок.

Переменные пакетов

Автономная транзакция создает новый контекст транзакции, но не новый сеанс. Поэтому любые переменные, находящиеся в области действия (доступные) родительской и автономной транзакции, будут в них идентичны, поскольку присвоение значений переменным не входит в транзакцию (нельзя вернуть переменной PL/SQL прежнее значение). Поэтому автономная транзакция может не только читать переменные состояния родительской транзакции, но и изменять их, и эти изменения будут видимы в родительской транзакции.

Это означает, что поскольку изменения значений переменных не фиксируются и не откатываются, эти изменения выпадают из области действия автономных транзакций и происходят так же, как и при отсутствии автономных транзакций. Чтобы продемонст-

рировать это на простом примере, я создам пакет с глобальной переменной. Родительская транзакция (наш сеанс) будет устанавливать этой переменной определенное значение, а в автономной транзакции оно будет изменяться. Это изменение скажется на родительской транзакции:

```
tkyte@TKYTE816> create or replace package global_variables
  2  as
  3          x number;
  4  end;
  5  /

Package created.

tkyte@TKYTE816> begin
  2         global_variables.x := 5;
  3  end;
  4  /

PL/SQL procedure successfully completed.

tkyteeTKYTE816> declare
  2         pragma autonomous_transaction;
  3  begin
  4         global_variables.x := 10;
  5          commit;
  6  end;
  7  /

PL/SQL procedure successfully completed.
tkyte@TKYTE816> set serveroutput on
tkyte@TKYTE816> exec dbms_output.put_line(global_variables.x);
10

PL/SQL procedure successfully completed.
```

Это изменение глобальной переменной автономной транзакцией останется в силе независимо от конечного результата (фиксации или отката) автономной транзакции.

Установки/параметры сеанса

Опять-таки, поскольку автономные транзакции создают новую транзакцию, а не новый сеанс, состояние сеанса в родительской транзакции будет таким же, как и в порожденной. Обе транзакции выполняются в одном сеансе, хотя и отдельно. Сеанс организуется при подключении приложения к базе данных. При выполнении автономной транзакции повторное подключение не выполняется — используется то же подключение и тот же сеанс. Поэтому любые изменения на уровне сеанса, выполненные в родительской транзакции, будут видимы в порожденной и, более того, если в порожденной транзакции выполняются изменения на уровне сеанса с помощью оператора **ALTER** SESSION, они повлияют и на родительскую транзакцию. Следует отметить, что оператор **SET TRANSACTION,** по определению работающий на уровне транзакции, влияет только на транзакцию, в которой выполнен. Так что, например, если в автономной транзакции выполнен оператор SET **TRANSACTION** USE **ROLLBACK SEGMENT,** то со-

ответствующий сегмент отката будет задан только для **автономной,** но не для родительской транзакции. Оператор **SET TRANSACTION ISOLATION LEVEL SERIALIZABLE,** выполненный в автономной транзакции, влияет только на эту транзакцию, а вот при выполнении оператора **ALTER SESSION SET ISOLATION_LEVEL=SERIALIZABLE** изменится и уровень изолированности следующей транзакции на уровне родительской. Кроме того, родительская транзакция, работающая в режиме **READ ONLY,** может вызвать автономную транзакцию, изменяющую базу данных. Автономная транзакция в этом случае может изменять данные.

Изменения в базе данных

Теперь переходим к самому интересному — изменениям в базе данных. Здесь происходящее несколько затуманивается. Изменения в базе данных, выполненные, но еще не зафиксированные родительской транзакцией, **невидимы** в автономных транзакциях. Изменения, выполненные и зафиксированные в родительской транзакции, всегда видимы порожденным транзакциям. Изменения, выполненные в автономной транзакции, могут **быть как видимы, так и невидимы** в родительской транзакции — это зависит от уровня ее изолированности.

Однако вот в чем неясность происходящего. Я вполне четко заявил, что изменения, выполненные в родительской транзакции, невидимы для порожденной, но это еще не все. Для курсора, отрытого в порожденной автономной транзакции, эти незафиксированные изменения невидимы. Но вот курсор, открытый в родительской транзакции, при выборе из него данных в порожденной, позволяет получить эти измененные данные. Следующий пример демонстрирует, о чем идет речь. Создадим новую таблицу **EMP** (для прежней мы создали всевозможные средства проверки), а затем напишем пакет, который ее изменяет и выдает содержимое. В этом пакете мы создадим глобальный курсор, выбирающий данные из таблицы **EMP.** В пакете будет одна процедура, оформленная как автономная транзакция. Она выбирает данные из курсора и выдает результаты. Сначала эта процедура проверяет, открыт ли курсор, и, если — нет, открывает его. Это позволит продемонстрировать разницу в получаемых результатах, в зависимости от того, где был открыт курсор. Результирующее множество курсора всегда согласовано на момент его открытия с учетом того, в какой транзакции он был открыт:

```
tkyte@TKYTE816> drop table emp;

Table dropped.

tkyte@TKYTE816> create table emp as select * from scott.emp;

Table created.

tkyte@TKYTE816> create or replace package my_pkg
  2  as
  3
  4          procedure run;
  5
  6  end;
  7  /

Package created.
```

```
tkyte@TKYTE816> create or replace package body my_pkg
  2  as
  3
  4
  5  cursor global_cursor is select ename from emp;
  6
  7
  8  procedure show_results
  9  is
 10          pragma autonomous_transaction;
 11          l_ename emp.ename%type;
 12  begin
 13          if (global_cursor%isopen)
 14          then
 15                  dbms_output.put_line('Еще НЕ открытый курсор');
 16          else
 17                  dbms_output.put_line('Уже открытый');
 18                  open global_cursor;
 19          end if;
 20
 21          loop
 22                  fetch global_cursor into l_ename;
 23                  exit when global_cursor%notfound;
 24                  dbms_output.put_line(l_ename);
 25          end loop;
 26          close global_cursor;
 27  end;
 28
 29
 30  procedure run
 31  is
 32  begin
 33         update emp set ename = 'x';
 34
 35          open global_cursor;
 36          show_results;
 37
 38          show_results;
 39
 40          rollback;
 41  end;
 42
 43  end;
 44  /

Package body created.

tkyte@TKYTE816> exec my_pkg.run
Еще НЕ открытый курсор
x

X
```

```
Уже открытый
SMITH

MILLER

PL/SQL procedure successfully completed.
```

Когда курсор открыт в родительской транзакции, в автономной транзакции можно получить незафиксированные строки — все значения **x**. Курсор, отрытый в автономной транзакции, с таким же успехом можно было открыть и в другом сеансе — для него эти незафиксированные данные недоступны. Мы видим данные в том состоянии, в каком они были до изменения.

Итак, этот пример показывает, как автономная транзакция будет реагировать на незафиксированные изменения в родительской транзакции при выполнении операторов **SELECT**. А будут ли в родительской транзакции видны изменения, произошедшие в автономной транзакции? Это будет зависеть от уровня изолированности родительской транзакции. При использовании стандартного уровня изолированности, **READ COMMITTED**, родительская транзакция сможет увидеть эти изменения. При использовании уровня изолированности **SERIALIZABLE** эти изменения не будут видны, хотя они выполнены в том же сеансе. Например:

```
tkyte@TKYTE816> create table t (msg varchar2(4000));

Table created.

tkyte@TKYTE816> create or replace procedure auto_proc
  2  as
  3          pragma autonomous_transaction;
  4  begin
  5          insert into t values ('A row for you');
  6          commit;
  7  end;
  8  /

Procedure created.

tkyte@TKYTE816> create or replace
  2  procedure proc(read_committed in boolean)
  3  as
  4  begin
  5          if (read_committed) then
  6                  set transaction isolation level read committed;
  7          else
  8                  set transaction isolation level serializable;
  9          end if;
 10
 11          auto_proc;
 12
 13          dbms_output.put_line('————————');
 14          for x in (select * from t) loop
 15                  dbms_output.put_line(x.msg);
 16          end loop;
 17          dbms_output.put_line('————————');
 18          commit;
```

```
 19  end;
 20  /
Procedure created.
tkyte@TKYTE816> exec proc(TRUE)
A row for you

PL/SQL procedure successfully completed.
tkyte@TKYTE816> delete from t;
1 row deleted.
tkyte@TKYTE816> commit;
Commit complete.
tkyte@TKYTE816> exec proc(FALSE)


PL/SQL procedure successfully completed.
```

Как видите, при выполнении процедуры в режиме **READ COMMITTED** зафиксированные изменения видны. При выполнении в режиме **SERIALIZABLE** изменения не видны. Дело в том, что изменения, выполненные в автономной транзакции, выполнены в другой транзакции, а уровень изолированности **SERIALIZABLE** требует учитывать только изменения, выполненные в данной транзакции (при этом уровне изолированности все обстоит так, как если бы транзакция была единственной — изменения, выполненные в других транзакциях, не видны).

Блокировки

В предыдущем разделе мы разобрались, что происходит при попытке чтения в порожденной автономной транзакции зафиксированных и незафиксированных изменений, выполненных родительской транзакцией, а также при чтении в родительской транзакции изменений, выполненных порожденной транзакцией. Теперь посмотрим, какие при этом устанавливаются блокировки.

Поскольку родительская и порожденная — это две абсолютно разные транзакции, они никак не могут совместно использовать блокировки. Если родительская транзакция заблокировала ресурс, который требуется заблокировать также порожденной автономной транзакции, произойдет взаимное блокирование в сеансе. Следующий пример демонстрирует эту проблему:

```
tkyte@TKYTE816> create or replace procedure child
  2  as
  3          pragma autonomous_transaction;
  4          l_ename emp.ename%type;
  5  begin
  6          select ename into l_ename
  7                    from emp
  8                   where ename = 'KING'
  9                     FOR UPDATE;
```

```
 10          commit;
 11  end;
 12  /

Procedure created.

tkyte@TKYTE816> create or replace procedure parent
  2  as
  3          l_ename emp.ename%type;
  4  begin
  5          select ename into l_ename
  6                    from emp
  7                   where ename = 'KING'
  8                     FOR UPDATE;
  9          child;
 10          commit;
 11  end;
 12  /

Procedure created.

tkyte@TKYTE816> exec parent
BEGIN parent; END;
*
ERROR at line 1:
ORA-00060: deadlock detected while waiting for resource
ORA-06512: at "TKYTE.CHILD", line 6
ORA-06512: at "TKYTE.PARENT", line 9
ORA-06512: at line 1
```

Необходимо проявлять осторожность и не допускать взаимного блокирования родительской и порожденной транзакций. Порожденная транзакция в этом случае всегда "проигрывает", и соответствующий ее оператор откатывается.

Завершение автономной транзакции

Для завершения автономной транзакции необходимо выполнять операторы **COMMIT** или **ROLLBACK**, или оператор ЯОД, который автоматически фиксирует транзакцию. Сама автономная транзакция начинается автоматически при выполнении изменения в базе данных, блокировании ресурсов или выполнении оператора управления транзакцией, такого как **SET TRANSACTION** или **SAVEPOINT**. Автономная транзакция должна быть явно завершена, прежде чем управление вернется в родительскую транзакцию (иначе выдается сообщение об ошибке). Отката до точки сохранения (**ROLLBACK TO SAVEPOINT**) недостаточно, даже если в результате незафиксированных изменений не остается, поскольку при этом не завершается транзакция.

Если автономная транзакция завершается нормально (а не путем распространения исключительной ситуации) и в ней не выполнен оператор **COMMIT** или **ROLLBACK**, выдается следующее сообщение об ошибке:

```
tkyte@TKYTE816> create or replace procedure child
  2  as
  3          pragma autonomous_transaction;
  4          l_ename emp.ename%type;
```

```
  5  begin
  6          select ename into l_ename
  7                    from emp
  8                   where ename =  'KING'
  9                     FOR UPDATE;
 10  end;
 11  /

Procedure created.

tkyte@TKYTE816> exec child
BEGIN child; END;
*
ERROR at line 1:
ORA-06519: active autonomous transaction detected and rolled back
ORA-06512: at "TKYTE.CHILD", line 6
ORA-06512: at line 1
```

Так что в случае автономной транзакции следует не только избегать взаимных блокировок с родительской, но и позаботиться о ее "чистом" завершении (чтобы предотвратить откат всех изменений).

Точки сохранения

В главе 4, посвященной транзакциям, я описывал точки сохранения и их влияние на выполняемые приложением транзакции. Точки сохранения действуют только в пределах текущей транзакции. Это означает, что нельзя откатить автономную транзакцию до точки сохранения, установленной в транзакции вызывающей подпрограммы. Этой точки сохранения нет в среде текущей автономной транзакции. Давайте посмотрим, что получится, если попытаться выполнить такой откат:

```
tkyte@TKYTE816> create or replace procedure child
  2  as
  3          pragma autonomous_transaction;
  4          l_ename emp.ename%type;
  5  begin
  6
  7         update emp set ename = 'y' where ename =  'BLAKE';
  8          rollback to Parent_Savepoint;
  9          commit;
 10  end;
 11  /

Procedure created.

tkyte@TKYTE816> create or replace procedure parent
  2  as
  3          l_ename emp.ename%type;
  4  begin
  5          savepoint  Parent_Savepoint;
  6         update emp set ename =  'x'  where ename =  'KING';
  7
  8          child;
  9          rollback;
```

```
 10 end;
 11 /

Procedure created.

tkyte@TKYTE816> exec parent
BEGIN parent; END;
*
ERROR at line 1:
ORA-01086: savepoint 'PARENT_SAVEPOINT' never established
ORA-06512: at "TKYTE.CHILD", line 8
ORA-06512: at "TKYTE.PARENT", line 8
ORA-06512: at line 1
```

Для автономной транзакции эта точка сохранения никогда не устанавливалась. Если удалить признак автономной транзакции из представленной выше процедуры **child** и повторно выполнить процедуру **parent,** все успешно сработает. Автономная транзакция не может изменять состояние родительской транзакции.

Это не означает, что в автономной транзакции нельзя использовать точки сохранения. Можно. Нужно только устанавливать **собственные** точки сохранения. Например, следующий код демонстрирует, что установленная в порожденной транзакции точка сохранения работает. Одно изменение, выполненное до точки сохранения, осталось, а другое — отменено, как и предполагалось:

```
tkyte@TKYTE816> create or replace procedure child
  2 as
  3          pragma autonomous_transaction;
  4          l_ename emp.ename%type;
  5  begin
  6
  7      update emp set ename = 'y' where ename = 'BLAKE';
  8       savepoint child_savepoint;
  9      update emp set ename = 'z' where ename = 'SMITH';
 10       rollback to child_savepoint;
 11       commit;
 12 end;
 13 /

Procedure created.

tkyte@TKYTE816> create or replace procedure parent
  2 as
  3          l_ename emp.ename%type;
  4  begin
  5          savepoint Parent_Savepoint;
  6         update emp set ename = 'x' where ename = 'KING';
  7
  8          child;
  9          commit;
 10 end;
 11 /

Procedure created.
```

```
tkyte@TKYTE816> select ename
  2    from emp
  3   where ename in ('x', 'y', 'z', 'BLAKE', 'SMITH', 'KING');

ENAME

SMITH
BLAKE
KING

tkyte@TKYTE816> exec parent

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select ename
  2    from emp
  3   where ename in ('x', 'y', 'z', 'BLAKE', 'SMITH', 'KING');

ENAME

SMITH
y
x
```

Проблемы

При использовании автономных транзакций имеется ряд нюансов, которые надо учитывать. В этом разделе мы рассмотрим их поочередно: какие возможности недоступны в автономных транзакциях, в каких средах автономные транзакции можно использовать, особенности, с которыми можно столкнуться при их использовании и другие подобные проблемы.

Невозможность использования в распределенных транзакциях

В текущих версиях (по крайней мере до Oracle 8.1.7) не допускается использование автономных транзакций в распределенной транзакции. Четкого сообщения об ошибке при этом не выдается. Во многих (но не во всех) случаях возникает внутренняя ошибка. В будущем планируется обеспечить надежную поддержку использования автономных транзакций в распределенных. Пока же, если используются связи базы данных, об автономных транзакциях лучше забыть.

Только в среде PL/SQL

Автономные транзакции доступны только в среде PL/SQL. Их можно перенести в Java и другие языки, вызвав соответствующую подпрограмму из блока PL/SQL, оформленного как автономная транзакция. Поэтому, если необходимо создать хранимую процедуру на языке Java, работающую как автономная транзакция, создают хранимую процедуру на PL/SQL, оформленную в виде автономной транзакции, и вызывают Java-процедуру из нее.

Откатывается вся транзакция

Если автономная транзакция завершается из-за ошибки, вследствие неперехваченной и необработанной исключительной ситуации, откатывается вся транзакция, а не только оператор, при выполнении которого произошла ошибка. Это означает, что при выполнении автономной транзакции вы получаете "все или ничего". Либо все изменения успешно фиксируются, **либо** возникает необработанная исключительная ситуация, и все **незафиксированные** изменения пропадают. Обратите внимание: **незафиксированные** изменения. В коде, оформленном как автономная транзакция, фиксация может выполняться многократно, и откатываются только незафиксированные изменения. Обычно, если при вызове процедуры возникает исключительная ситуация, которая перехватывается и обрабатывается в вызывающем коде, незафиксированные изменения остаются, но не в случае автономной транзакции. Например:

```
tkyte@TKYTE816> create table t (msg varchar2(25));

Table created.

tkyte@TKYTE816> create or replace procedure auto_proc
  2 as
  3     pragma AUTONOMOUS_TRANSACTION;
  4     x number;
  5 begin
  6     insert into t values ('AutoProc');
  7     x := 'a'; — При выполнении этого оператора произойдет ошибка
  8     commit;
  9 end;
 10 /

Procedure created.

tkyte@TKYTE816> create or replace procedure Regular_Proc
  2 as
  3     x number;
  4 begin
  5     insert into t values ('RegularProc');
  6     x := 'a'; — При выполнении этого оператора произойдет ошибка
  7     commit;
  8 end;
  9 /

Procedure created.

tkyte@TKYTE816> set serveroutput on

tkyte@TKYTE816> begin
  2     insert into t values ('Anonymous');
  3     auto_proc;
  4 exception
  5     when others then
  6                  dbms_output.put_line('Перехвачена ошибка:');
  7                 dbms_output.put_line(sqlerrm);
  8         commit;
  9 end;
 10 /
```

```
Перехвачена ошибка:
ORA-06502: PL/SQL: numeric or value error: character to number conversion
           error

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select * from t;

MSG

Anonymous
```

Сохранились только данные, вставленные в анонимном блоке. Сравните это с поведением "обычного" блока:

```
tkyte@TKYTE816> delete from t;

1 row deleted.

tkyte@TKYTE816> commit;

Commit complete.

tkyte@TKYTE816> begin
  2      insert into t values ('Anonymous');
  3      regular_proc;
  4  exception
  5      when others then
  6                   dbms_output.put_line('Перехвачена ошибка:');
  7                  dbms_output.put_line(sqlerrm);
  8          commit;
  9  end;
 10  /
Перехвачена ошибка:
ORA-06502: PL/SQL: numeric or value error: character to number conversion
->error

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select * from t;

MSG

Anonymous
RegularProc
```

В данном случае, поскольку ошибка перехвачена и обработана, в таблице остались строки, вставленные как анонимным блоком, так и завершившейся ошибкой процедурой.

Это означает, что нельзя просто добавить прагму **AUTONOMOUS_TRANSACTION** в существующие хранимые процедуры в надежде, что они будут работать как прежде. Могут появиться существенные отличия.

Временные таблицы уровня транзакции

При использовании временных (**GLOBAL TEMPORARY**) таблиц необходимо учитывать, что временные таблицы уровня транзакции нельзя одновременно использовать в нескольких транзакциях в одном сеансе. Временные таблицы управляются на уровне сеанса и при создании их на уровне транзакции (с удалением всех строк при фиксации)

могут использоваться только в родительской или только в порожденной транзакции, но не в обеих. Так, следующий пример показывает, что автономная транзакция, пытающаяся читать или изменять временную таблицу **уровня транзакции,** уже использующуюся в сеансе, не срабатывает:

```
tkyte@TKYTE816> create global temporary table temp
  2 (x int)
  3 on commit delete rows
  4 /

Table created.

tkyte@TKYTE816> create or replace procedure auto_proc1
  2 as
  3         pragma autonomous_transaction;
  4 begin
  5     insert into temp values (1);
  6      commit;
  7 end;
  8 /

Procedure created.

tkyte@TKYTE816> create or replace procedure auto_proc2
  2 as
  3         pragma autonomous_transaction;
  4 begin
  5         for x in (select * from temp)
  6         loop
  7                 null;
  8         end loop;
  9      commit;
 10 end;
 11 /

Procedure created.

tkyte@TKYTE816> insert into temp values (2);

1 row created.

tkyte@TKYTE816> exec auto_proc1;
BEGIN auto_proc1; END;
*
ERROR at line 1:
ORA-14450: attempt to access a transactional temp table already in use
ORA-06512: at "TKYTE.AUTO_PROC1", line 5
ORA-06512: at line 1


tkyte@TKYTE816> exec auto_proc2;
BEGIN auto_proc2; END;
*
ERROR at line 1:
ORA-14450: attempt to access a transactional temp table already in use
ORA-06512: at "TKYTE.AUTO_PROC2", line 5
ORA-06512: at line 1
```

Именно это сообщение об ошибке вы и получите при попытке использовать одну и ту же временную таблицу в обеих транзакциях. Следует отметить, что это происходит только с одновременными транзакциями в **одном сеансе.** Несколько одновременно выполняющихся транзакций, если только каждая из них выполняется в отдельном сеансе, могут обращаться к временным таблицам уровня транзакции.

Изменяющиеся таблицы

Казалось бы, автономные транзакции позволяют решить все проблемы изменяющихся таблиц. Эти решения, однако, могут стать началом **новых** логических проблем.

Предположим, необходимо обеспечить выполнение правила, по которому максимальная зарплата сотрудника не может более чем вдвое превышать среднюю зарплату сотрудников соответствующего отдела. Можно начать с процедуры и триггера примерно следующего вида:

```
tkyte@TKYTE816> create or replace
  2  procedure sal_check(p_deptno in number)
  3  is
  4          avg_sal number;
  5          max_sal number;
  6  begin
  7          select avg(sal), max(sal)
  8            into avg_sal, max_sal
  9             from emp
 10       where deptno = p_deptno;
 11
 12          if (max_sal/2 > avg_sal)
 13          then
 14                  raise_application_error(-20001,'Rule violated');
 15         end if;
 16 end;
 17 /

Procedure created.

tkyte@TKYTE816> create or replace trigger sal_trigger
  2  after insert or update or delete on emp
  3  for each row
  4  begin
  5          if (inserting or updating) then
  6                  sal_check(:new.deptno);
  7          end if;
  8
  9          if (updating or deleting) then
 10                  sal_check(:old.deptno);
 11         end if;
 12 end;
 13 /

Trigger created.

tkyte@TKYTE816>
tkyte@TKYTE816> update emp set sal = sal*1.1;
```

```
update emp set sal = sal*1.1

ERROR at line 1:
ORA-04091: table TKYTE.EMP is mutating, trigger/function may not see it
ORA-06512: at "TKYTE.SAL_CHECK", line 6
ORA-06512: at "TKYTE.SAL_TRIGGER", line 3
ORA-04088: error during execution of trigger "TKYTE.SAL_TRIGGER'
```

Не слишком удачно. Мы сразу же столкнулись с ошибкой изменяющейся таблицы, поскольку нельзя читать таблицу в процессе ее изменения. Сразу приходит в голову мысль: раз таблица изменяется, надо использовать автономную транзакцию. Это и делается:

```
tkyte@TKYTE816> create or replace
  2  procedure sal_check(p_deptno in number)
  3  is
  4          pragma autonomous_transaction;
  5          avg_sal number;
  6          max_sal number;
  7  begin


Procedure created.
```

Кажется, что проблема решена:

```
tkyte@TKYTE816> update emp set sal = sal*1.1;

14 rows updated.

tkyte@TKYTE816> commit;

Commit complete.
```

При более детальном рассмотрении, однако, оказывается, что эта идея принципиально ошибочна. В ходе тестирования обнаруживается, что велика вероятность следующего:

```
tkyte@TKYTE816> update emp set sal = 99999.99 where ename = 'WARD';

1 row updated.

tkyte@TKYTE816> commit;

Commit complete.

tkyte@TKYTE816> exec sal_check(30);
BEGIN sal_check(30); END;
*

ERROR at line 1:
ORA-20001: Rule violated
ORA-06512: at "TKYTE.SAL_CHECK", line 14
ORA-06512: at line 1
```

Я изменил запись служащего **WARD**, установив ему очень большую зарплату; **WARD** работает в отделе 30, и его зарплата теперь намного превышает среднюю зарплату по этому отделу. Триггер этого не выявил, но постфактум, выполнив тот же код, что выполняет триггер, мы обнаруживаем нарушение правила. Почему? Потому что в автоном-

ной транзакции невидимы выполняемые нами изменения. Поэтому такое увеличение зарплаты и проходит: процедура проверяет таблицу по состоянию до начала этого изменения! С нарушением столкнется следующий невезучий пользователь (как мы продемонстрировали, искусственно вызвав процедуру SAL_CHECK).

При любом использовании автономной транзакции во избежание проблемы изменяющейся таблицы убедитесь, что вы поступаете правильно. В разделе *"Проверка, записи которой не могут быть отменены"* я использовал автономную транзакцию "безопасным" способом. Логика работы триггера не нарушается из-за того, что таблица в нем видна в состоянии до начала транзакции. В представленном выше примере триггер от этого существенно пострадал. Необходимо быть особенно внимательным и проверять корректность каждого триггера, оформленного как автономная транзакция.

Ошибки, которые могут произойти

При работе с автономными транзакциями может произойти несколько ошибок. Для полноты изложения соответствующие сообщения представлены и прокомментированы ниже, но большинство них мы уже встречали в примерах.

В этом разделе текст сообщения об ошибке приведен так, как он выдается СУБД Oracle версии 8.1.6.0.0 при установке русского языка для сообщений. Обратите внимание на несоответствие терминологии, предлагаемой компанией Oracle. В примерах оставлены сообщения на английском языке. — *Прим. научн. ред.*

ORA-06519: выполнен откат назад для незавершенной автономной транзакции

```
// * Причина: Перед выходом из автономного PL/SQL-блока все начатые в нем
//           автономные транзакции должны быть завершены (зафиксированы
//           или отменены). В противном случае активная автономная
//           транзакция неявно откатывается и выдается это сообщение об
//             ошибке.
// * Действие: Убедитесь, что перед выходом из автономного PL/SQL-блока
//           все активные автономные транзакции явно зафиксированы или
//             отменены.
```

Это сообщение об ошибке выдается каждый раз при выходе из автономной транзакции, если перед этим не позаботились о ее явной фиксации или откате. При этом автономная транзакция откатывается, а исключительная ситуация распространяется в среду, откуда автономная транзакция была вызвана. Чтобы избавиться от этой проблемы, следует всегда обеспечивать фиксацию или откат транзакции для всех вариантов завершения PL/SQL-блока, оформленного как автономная транзакция. Эта ошибка всегда связана с логической ошибкой в коде.

ORA-14450: попытка доступа к уже используемой временной таблице транзакций

```
// * Причина: Выполнена попытка доступа к временной таблице уровня
//           транзакции, данные в которую поместила другая транзакция,
//           одновременно выполняющаяся в том же сеансе.
```

```
// * Действие: Не пытайтесь обращаться к временной таблице, пока
//             одновременно выполняющаяся транзакция не будет
//             зафиксирована или отменена.
```

Как было продемонстрировано ранее, глобальная временная таблица, созданная с опцией ON COMMIT DELETE ROWS, может использоваться только одной транзакцией в сеансе. Необходимо не допускать использования одной временной таблицы в родительской и порожденной транзакции.

ORA-00060: взаимная блокировка при ожидании ресурса

```
// * Причина: Произошла взаимная блокировка транзакций при ожидании
//            ресурса.
// * Действие: Определите по трассировочному файлу, какие транзакции и
//             ресурсы стали причиной взаимной блокировки. При
//             необходимости повторите транзакцию.
```

Эта ошибка не связана непосредственно с использованием автономных транзакций, но я представил ее здесь потому, что при использовании автономных транзакций повышается вероятность ее возникновения. Поскольку родительская транзакция приостанавливается на время выполнения порожденной транзакции, и не может продолжить работу до ее завершения, взаимная блокировка, не возникающая при одновременной работе двух сеансов, возникает при использовании автономной транзакции. Она произойдет при попытке изменить одни и те же данные в двух отдельных транзакциях одного сеанса. Необходимо проверять, не пытается ли порожденная транзакция заблокировать ресурсы, уже заблокированные родительской транзакцией.

Резюме

В этой главе мы детально изучили возможности автономных транзакций. Вы увидели, как их можно использовать для создания более модульного и безопасного кода. Вы научились выполнять с их помощью невозможные до этого действия (например, выполнять операторы ЯОД в триггере или вызывать в операторе **SELECT** хранимую функцию независимо от того, изменяет ли она базу данных). Я объяснил, почему неразумно предполагать, что функция, вызываемая в SQL-операторе, будет выполнена определенное количество раз, и поэтому надо быть особенно осторожным при изменении базы данных в таких функциях. Вы узнали, как с помощью автономных транзакций избежать проблемы изменяющейся таблицы, и что они могут привести к ошибочному результату при некорректном использовании для решения этой проблемы.

Автономные транзакции — мощное средство, которое сервер Oracle использует уже многие годы для выполнения рекурсивных SQL-операторов. Теперь их можно использовать и в приложениях. Прежде чем использовать это средство, надо хорошо понимать, как выполняются транзакции, когда они начинаются и когда заканчиваются, поскольку могут возникать различные побочные эффекты. Например, сеанс может заблокировать сам себя, родительская транзакция может видеть или не видеть результаты порожденной автономной, порожденная автономная транзакция не видит незафиксированные результаты родительской и т.д.

16

Динамический SQL

Обычно при разработке программ все используемые в них SQL-операторы явно записываются в исходном коде. Такой вариант использования SQL-операторов обычно называют *статический SQL*. Многие полезные программы, однако, до момента запуска не "знают", какие именно SQL-операторы будут выполняться. Именно так и появляется *динамический SQL* — программа при запуске выполняет SQL-операторы, неизвестные во время компиляции. Возможно, программа генерирует запросы по ходу работы на основе введенных пользователем условий; возможно, это специализированная программа загрузки данных. Утилита SQL*Plus — прекрасный пример такого рода программы, как и любое другое средство выполнения произвольных запросов или генерации отчетов. Утилита SQL*Plus позволяет выполнить любой SQL-оператор и показать результаты его выполнения* хотя при ее компиляции операторы, которые выполняет пользователь, определенно не были известны.

В этой главе мы обсудим, когда возникает необходимость использовать динамический SQL в программах и когда его имеет смысл применять. Мы сосредоточимся на использовании динамического SQL в программах на языке PL/SQL, поскольку именно в этой среде большинство разработчиков и используют динамический SQL в предварительно компилируемом формате. Поскольку использование динамического SQL — единственный способ выполнить SQL-операторы в программах на языке Java через интерфейс JDBC (выполнить динамический SQL в среде прекомпилятора SQLJ можно только через интерфейс JDBC) и на языке C при использовании библиотеки OCI, не имеет смысла обсуждать эти среды в данном контексте. В этих средах есть только динамический SQL; статический SQL вообще не поддерживается, так что там просто нет выбора. Мы же в данной главе:

- рассмотрим последствия использования динамического или статического SQL;
- разберемся, как использовать динамический SQL в программах с помощью средств стандартного пакета **DBMS_SQL;**
- изучим возможности *встроенного* (native) динамического SQL;
- рассмотрим ряд проблем, с которыми можно столкнуться при использовании динамического SQL в приложениях, в частности нарушение цепочки зависимостей, уязвимость кода и трудности при его настройке.

Для выполнения всех примеров динамического SQL в этой главе необходим сервер Oracle 8.1.5 или более новых версий. Встроенный динамический SQL появился именно в этой версии и является одной из важнейших возможностей всех последующих версий. Для выполнения большинства примеров, в которых используются средства пакета **DBMS_SQL,** достаточно сервера Oracle версии 7.1 или более новых версий (правда, функции, обрабатывающие массивы, появились в пакете **DBMS_SQL** в версии 8.0).

Сравнение динамического и статического SQL

Использование динамического SQL — естественная возможность работать с базой данных через функциональный интерфейс, такой как ODBC, JDBC и OCI. Статический SQL обычно принято использовать в средах с предварительной компиляцией кода, таких как Pro*C, SQLJ и PL/SQL (я не оговорился: компилятор PL/SQL можно рассматривать как прекомпилятор). При работе через функциональный интерфейс поддерживается **только** динамический SQL. Программист создает запрос в виде строки, а затем эта строка анализируется, связываются входящие в нее переменные, запрос выполняется, при необходимости выбираются строки из результирующего множества через курсор и, наконец, соответствующий курсор закрывается. В среде статического SQL эти действия выполняются автоматически. Для сравнения создадим две выполняющие одинаковые действия PL/SQL-процедуры: одну с — использованием динамического SQL, а вторую — с использованием статического. Вот версия на основе динамического SQL:

```
scott@TKYTE816> create or replace procedure DynEmpProc(p_job in varchar2)
  2  as
  3     type refcursor is ref cursor;
  4
  5     -- При использовании динамического SQL необходимо
  6     -- создать хост-переменные и выделить ресурсы.
  7     l_cursor   refcursor;
  8     l_ename    emp.ename%type;
  9  begin
 10
 11     -- Начинаем с анализа запроса
 12     open l_cursor for
 13     'select ename
 14        from emp
 15      where job = :x' USING in p_job;
```

```
 16
 17      loop
 18          -- и явно ВЫБИРАЕМ данные через курсор.
 19          fetch l_cursor into l_ename;
 20
 21          -- Необходимо самостоятельно обрабатывать ошибки
 22          -- и делать выборку
 23          exit when l_cursor%notfound;
 24
 25          dbms_output.put_line(l_ename);
 26      end loop;
 27
 28       -- Не забываем освободить ресурсы
 29      close l_cursor;
 30  exception
 31      when others then
 32          -- а также перехватить и обработать все ошибки,
 33          -- чтобы не допустить утечки ресурсов
 34          -- при возникновении ошибок.
 35          if (l_cursor%isopen)
 36          then
 37              close l_cursor;
 38          end if;
 39          RAISE;
 40  end;
 41  /
```

```
Procedure created.
```

А вот что мы имеем в случае статического SQL:

```
scott@TKYTE816> create or replace procedure StaticEmpProc(p_job in varchar2)
  2  as
  3  begin
  4      for x in (select ename from emp where job = p_job)
  5      loop
  6          dbms_output.put_line(x.ename);
  7      end loop;
  8  end;
  9  /

Procedure created.
```

Эти две процедуры делают то же самое:

```
scott@TKYTE816> set serveroutput on size 1000000
scott@TKYTE816> exec DynEmpProc('CLERK')
SMITH
ADAMS
JAMES
MILLER

PL/SQL procedure successfully completed.

scott@TKYTE816> exec StaticEmpProc('CLERK')
```

```
SMITH
ADAMS
JAMBS
MILLER

PL/SQL procedure successfully completed.
```

Понятно, однако, что версия с динамическим SQL требует от разработчика написания гораздо большего объема кода. По опыту знаю: статический SQL обеспечивает более высокую производительность труда программиста при написании кода (приложения разрабатываются быстрее), но динамический SQL обеспечивает большую гибкость при выполнении (программа в ходе работы может делать то, что не внесено в ее код явно). Кроме того, статический SQL (особенно в среде PL/SQL) будет выполняться намного эффективнее, чем динамический. Используя статический SQL, PL/SQL-машина при обработке одной строки интерпретируемого кода может сделать то, на что потребуется пять или шесть строк интерпретируемого кода с динамическим SQL. Поэтому я использую статический SQL где только возможно и применяю динамический, только если по-другому задачу решить нельзя. Оба они эффективны, ни один не имеет принципиальных преимуществ перед другим, и оба имеют свои специфические возможности и средства повышения производительности.

Когда использовать динамический SQL?

Многие задачи требуют использования динамического SQL в PL/SQL. Вот лишь некоторые из них.

- Разработка обобщенных процедур, выполняющих стандартные действия вроде выгрузки данных в файлы. В главе 9 был представлен пример такой процедуры.
- Разработка универсальных процедур загрузки данных в не известные заранее таблицы. Мы рассмотрим использование динамического SQL для загрузки данных в таблицу.
- Динамический вызов других PL/SQL-процедур во время выполнения. Эта тема затрагивается в главе 23. Здесь мы рассмотрим ее более детально.
- Генерация условий (например, конструкции **WHERE)** в процессе работы на основе введенных пользователем данных. Это, пожалуй, основная причина использования динамического SQL большинством разработчиков. Я покажу в этой главе, как это надо (и как не надо!) делать.
- Выполнение операторов ЯОД. Поскольку PL/SQL не разрешает включать статические операторы ЯОД в код приложения, остается использовать динамический SQL. Это позволит выполнять операторы, начинающиеся с ключевых слов **CREATE, ALTER, GRANT, DROP** и т.п.

Решаться перечисленные задачи будут с помощью двух средств языка PL/SQL. Сначала мы рассмотрим использование стандартного пакета **DBMS_SQL**. Этот пакет существует уже достаточно давно, он появился в версии 7.1. Пакет обеспечивает процедурный метод выполнения динамического SQL, аналогичный использованию

функциональных интерфейсов (таких как JDBC или ODBC). Затем поговорим о встроенном динамическим SQL (который реализуется в PL/SQL оператором **EXECUTE IMMEDIATE).** Это декларативный способ выполнения динамического SQL в языке PL/SQL и в большинстве случаев он синтаксически намного проще, чем использование пакета **DBMS_SQL;** кроме того, он обеспечивает более высокую производительность.

Учтите, что **многие** подпрограммы пакета **DBMS_SQL** по-прежнему являются жизненно важными и активно используются в PL/SQL. Мы сравним два метода и попытаемся четко сформулировать, когда имеет смысл использовать каждый из них. Как только стало понятно, что необходимо использовать динамический SQL (статический SQL — лучший выбор в большинстве случаев), придется выбирать реализацию на основе пакета **DBMS_SQL** или встроенного динамического SQL.

Пакет **DBMS_SQL** необходимо использовать в следующих случаях.

- Если заранее не известно количество или типы столбцов, с которыми придется работать. Пакет **DBMS_SQL** включает процедуры для описания результирующего множества. Встроенный динамический SQL не позволяет получить такое описание. При использовании встроенного динамического SQL необходимо знать характеристики результирующего множества при компиляции, если результаты необходимо обрабатывать в PL/SQL.
- Если заранее не известно количество или типы связываемых переменных, с которыми придется работать. Пакет **DBMS_SQL** по ходу выполнения позволяет привязать с помощью процедур входные переменные к операторам. Встроенный динамический SQL требует учета количества и типов связываемых переменных на этапе компиляции (я приведу интересный способ решения этой проблемы).
- Когда необходимо выбирать или вставлять тысячи строк и можно использовать обработку массивов. Пакет **DBMS_SQL** поддерживает обработку массивов — возможность выбрать N строк за раз, одним вызовом. Встроенный динамический SQL обычно не позволяет этого сделать, но это ограничение можно обойти, как будет показано далее.
- Если в сеансе многократно выполняется один и тот же оператор. Пакет **DBMS_SQL** позволяет один раз разобрать оператор, а затем выполнять его многократно. При использовании встроенного динамического SQL мягкий разбор будет осуществляться при каждом выполнении. В главе 10 было показано, почему такие дополнительные повторные разборы нежелательны.

Встроенный динамический SQL имеет смысл использовать в следующих случаях.

- Когда количество и типы столбцов, с которыми придется работать, заранее известны.
- Когда заранее известно количество и типы связываемых переменных. (Можно также использовать *контексты приложений,* чтобы с помощью более простого встроенного динамического SQL выполнять операторы с заранее неизвестным количеством или типами связываемых переменных.)
- Когда необходимо выполнять операторы ЯОД.

- Если динамически формируемые операторы будут выполняться лишь несколько раз (оптимальный вариант — однократно).

Использование динамического SQL

Я рассмотрю основные шаги при использовании как стандартного пакета **DBMS_SQL,** так и возможностей встроенного динамического SQL.

Пакет DBMS_SQL

DBMS_SQL — это стандартный встроенный пакет, поставляемый вместе с сервером. Стандартно он устанавливается в схеме пользователя SYS, а привилегия для его выполнения предоставляется роли **PUBLIC.** Это означает, что не должно быть никаких проблем с доступом к нему или созданием хранимых объектов, ссылающихся на его процедуры, — никаких дополнительных или специальных привилегий для этого предоставлять не надо. Одним из положительных свойств пакета является доступность соответствующей документации. Если при использовании DBMS_SQL необходимо вспомнить ту или иную особенность, можно просто выполнить следующий сценарий:

```
scott@TKYTE816> set pagesize 30
scott@TKYTE816> set pause on
scott@TKYTE816> prompt Не забудьте нажать ENTER, чтобы получить результа
Не забудьте нажать ENTER, чтобы получить результат

scott@TKYTE816> select text
  2    from all_source
  3   where name = 'DBMS_SQL'
  4     and type = 'PACKAGE'
  5   order by line
  6  /

TEXT

package dbms_sql is

--   OVERVIEW

-- This package provides a means to use dynamic SQL to access the
database.


-- RULES AND LIMITATIONS
```

*Если необходимо узнать возможности или просмотреть примеры, используйте этот прием для всех стандартных пакетов **DBMS_** или **UTL_**.*

Пакет **DBMS_SQL** реализует процедурный подход к использованию динамического SQL. Этот подход сходен с тем, который используется в других языках (например,

при программировании на Java с использованием **JDBC** или на C с использованием библиотеки OCI) В общем случае, процесс, использующий пакет **DBMS_SQL,** будет иметь следующую структуру.

- Вызов **OPEN_CURSOR** для получения дескриптора курсора.
- Вызов **PARSE** для анализа оператора. Один и тот же дескриптор курсора можно использовать для обработки нескольких операторов. В каждый момент времени, однако, обрабатывается только один оператор.
- Вызов **BIND_VARIABLE** или **BIND_ARRAY** для передачи входных данных оператору.
- Если обрабатывается запрос (оператор **SELECT),** необходимо вызвать процедуру **DEFINE_COLUMN** или **DEFINE_ARRAY,** чтобы указать серверу Oracle, как передавать результаты (как массивы или как скалярные величины и какой тип данных при этом использовать).
- Вызов **EXECUTE** для выполнения оператора.
- Если выполняется запрос, необходимо вызвать **FETCH_ROWS** для выборки данных. Для получения данных по порядковому месту в списке выбора используется вызов **COLUMN_VALUE.**
- Если же выполняется блок кода PL/SQL или оператор ЯМД с конструкцией **RETURN,** можно вызвать процедуру **VARIABLE_VALUE** для получения результатов (параметров типа **OUT)** из блока по имени.
- Вызов **CLOSE_CURSOR.**

В следующем псевдокоде продемонстрирована последовательность шагов для динамического выполнения запроса:

```
1) Открыть курсор
2) Проанализировать оператор
3) При необходимости получить описание оператора, чтобы выяснить
     количество и типы возвращаемых столбцов
4) Выполнить цикл по i по связываемым переменным (входным)
     Связать i-ую входную переменную с оператором
5) Выполнить цикл по i по возвращаемым столбцам
     Определить i-ый столбец, сообщив серверу Oracle тип переменной, в
     которую будут выбираться данные
6) Выполнить оператор
7) Выполнять цикл пока удается выбрать строку
8)  Выполнить цикл по i по возвращаемым столбцам
     Получить значение i-го столбца строки с помощью column_value
   Конец цикла по строкам
9) Закрыть курсор
```

Для выполнения PL/SQL-блока или оператора ЯМД используется следующий псевдокод:

```
1) Открыть курсор
2) Проанализировать оператор
3) Выполнить цикл по i по связываемым переменным (входным и выходным)
```

```
     Связать i-ую переменную с оператором
4) Выполнить оператор
5) Выполнить цикл по i по выходным связываемым переменным
     Получить значение i-й выходной переменной с помощью variable_value
6) Закрыть курсор
```

Наконец, при выполнении операторов ЯОД (в которых нельзя использовать связываемые переменные), PL/SQL-блоков или операторов ЯМД, в которых нет связываемых переменных, представленный выше алгоритм упрощается (хотя, для этого типа операторов я всегда предпочитаю использовать не пакет **DBMS_SQL,** а встроенный динамический SQL):

```
1) Открыть курсор
2) Проанализировать оператор
3) Выполнить оператор
4) Закрыть курсор
```

Рассмотрим пример использования пакета DBMS_SQL для выполнения запроса, подсчитывающего количество строк в таблице базы данных, к которой пользователь имеет доступ:

```
scott@TKYTE816> create or replace
  2 function get_row_cnts(p_tname in varchar2) return number
  3 as
  4     l_theCursor     integer;
  5     l_columnValue   number  default NULL;
  6     l_status        integer;
  7 begin
  8
  9      -- Шаг 1, открыть курсор.
 10    1 theCursor := dbms_sql.open_cursor;
```

Мы начинаем блок с обработчиком исключительных ситуаций. Если по ходу работы этого блока возникает ошибка, необходимо закрыть только что открытый курсор в обработчике исключительных ситуаций, чтобы предотвратить "утечку курсоров", когда дескриптор курсора теряется при распространении исключительной ситуации за пределы функции.

```
 11     begin
 12
 13         -- Шаг 2, проанализировать запрос.
 14          dbms_sql.parse(c           => l_theCursor,
 15                         statement     => 'select count(*) from ' ||
->p_tname,
 16                           language_flag => dbms_sql.native);
 17
```

Обратите внимание, что параметр **language_flag** получает значение одной из констант пакета **DBMS_SQL.NATIVE**. Это вызывает анализ запроса по правилам сервера, выполняющего код. Можно также задать значения **DBMS_SQL.V6** или **DBMS_SQL.V7**. Я всегда использую значение **NATIVE**.

Шаги 3 и 4 из представленного ранее псевдокода нам не нужны, поскольку результаты **известны** и никаких связываемых переменных в этом примере нет.

```
18      -- Шаг 5, убедиться, что запрос возвращает данные типа NUMBER.
19         dbms_sql.define_column (c        => l_theCursor,
20                                 position => 1,
21                                 column   => l_columnValue);
22
```

Процедура **DEFINE_COLUMN** — перегруженная, так что компилятор сам определяет, когда надо вызывать версию для типа **NUMBER**, а когда — для **DATE** или **VARCHAR**.

```
23          -- Шаг 6, выполнить оператор.
24         l_status := dbms_sql.execute(l_theCursor);
25
```

Если бы выполнялся оператор **ЯМД**, переменная **L_STATUS** получила бы значение, равное количеству возвращенных строк. Для оператора **SELECT** возвращаемое значение несущественно.

```
26          -- Шаг 7, выбрать строки.
27          if (dbms_sql.fetch_rows(c => l_theCursor) > 0)
28          then
29              -- Шаг 8, получить значения из очередной строки.
30              dbms_sql.column_value(c        => l_theCursor,
31                                    position => 1,
32                                    value    => l_columnValue);
33        end if;
34
35          -- Шаг 9, закрыть курсор.
36      dbms_sql.close_cursor(c     => l_theCursor);
37          return l_columnValue;
38      exception
39          when others then
40             dbms_output.put_line('==> ' || sqlerrm);
41              dbms_sql.close_cursor(c => l_theCursor);
42              RAISE;
43     end;
44 end;
45 /

Function created.

scott@TKYTE816> set serveroutput on
scott@TKYTE816> begin
  2     dbms_output.put_line('Emp has this many rows ' ||
  3                          get_row_cnts('emp'));
  4 end;
  5 /
Emp has this many rows 14

PL/SQL procedure successfully completed.

scott@TKYTE816> begin
  2     dbms_output.put_line('Not a table has this many rows ' ||
```

```
  3                              get_row_cnts('NOT_A_TABLE')>;
  4  end;
  5  /
===> ORA-00942: table or view does not exist
begin

ERROR at line 1:
ORA-00942: table or view does not exist
ORA-06512: at "SCOTT.GET_ROW_CNTS", line 60
ORA-06512: at line 2
```

Рассмотренный пример начинается созданием курсора с помощью вызова **DBMS_SQL.OPEN_CURSOR.** Следует отметить, что это специфический курсор **DBMS_SQL** — его нельзя передать для выборки данных в приложение на Visual Basic или использовать в качестве PL/SQL-курсора. Для выборки данных с помощью этого курсора необходимо использовать подпрограммы пакета **DBMS_SQL.** Затем мы проанализировали запрос **SELECT COUNT(*) FROM TABLE,** где значение **TABLE** передается при вызове во время выполнения — оно просто конкатенируется со строкой запроса. Приходится "вклеивать" имя таблицы в запрос, поскольку связываемые переменные **нельзя** использовать в качестве идентификатора (имени таблицы или имени столбца, например). После анализа запроса мы вызвали **DBMS_SQL.DEFINE_COLUMN,** чтобы указать, что первый (и единственный в данном случае) столбец в списке **SELECT** должен интерпретироваться при выборке как тип **NUMBER.** То, что мы хотим выбирать данные именно этого типа, явно не указано — процедура **DBMS_SQL.DEFINE_COLUMN** перегружена и имеет несколько версий для данных типа **VARCHAR, NUMBER, DATE, BLOB, CLOB** и так далее. Тип возвращаемого значения определяется по типу переменной, в которую он помещается. Поскольку переменная **L_COLUMNVALUE** в рассмотренном примере имеет тип **NUMBER,** вызывается версия процедуры **DEFINE_COLUMN** для чисел. Затем мы вызываем **DBMS_SQL.EXECUTE.** Если бы выполнялся оператор **INSERT, UPDATE** или **DELETE,** функция **EXECUTE** вернула бы количество затронутых строк. В случае запроса возвращаемое значение функции не определено, и его можно проигнорировать. После выполнения оператора вызывается функция **DBMS_SQL.FETCH_ROWS.** Функция **FETCH_ROWS** возвращает количество фактически выбранных строк. В нашем случае, поскольку связывались скалярные переменные (не массивы), функция **FETCH_ROWS** будет возвращать 1 до тех пор, пока не исчерпаются данные, — тогда она вернет 0. При извлечении каждой строки мы вызываем **DBMS_SQL.COLUMN_VALUE** для каждого столбца в списке выбора, чтобы получить его значение. Наконец, мы завершаем выполнение функции, закрывая курсор с помощью вызова **DBMS_SQL.CLOSE_CURSOR.**

Теперь рассмотрим, как использовать пакет **DBMS_SQL** для обработки динамически формируемых параметризованных PL/SQL-блоков или операторов ЯМД. Я часто использую такое динамическое формирование, например, при загрузке данных из файлов операционной системы с помощью пакета **UTL_FILE** (он позволяет читать текстовые файлы в PL/SQL). Пример подобного рода утилиты был представлен в главе 9. Там мы использовали пакет **DBMS_SQL** для динамического построения операторов **INSERT,** в которых количество столбцов становится известным только при выполнении и меня-

ется от вызова к вызову. Нельзя использовать встроенный динамический SQL для загрузки в таблицу произвольного количества столбцов, поскольку для этого уже на этапе компиляции необходимо точно знать количество связываемых переменных. Следующий пример создан специально, чтобы показать особенности использования подпрограмм пакета DBMS_SQL при работе с блоками PL/SQL и операторами ЯМД (это пример проще реализовать с помощью встроенного динамического SQL, поскольку в этом случае количество связываемых переменных известно во время компиляции):

```
scott@TKYTE816> create or replace
  2  function update_row(p_owner   in varchar2,
  3                      p_newDname in varchar2,
  4                      p_newLoc   in varchar2,
  5                      p_deptno   in varchar2,
  6                      p_rowid    out varchar2)
  7  return number
  8  is
  9      l_theCursor    integer;
 10     l_columnValue  number  default NULL;
 11      l_status       integer;
 12      l_update       long;
 13  begin
 14     l_update := 'update ' || p_owner || '.dept
 15                    set dname = :bv1, loc = :bv2
 16                  where deptno = to_number(:pk)
 17              returning rowid into :out';
 18
 19      -- Шаг 1, открыть курсор.
 20     l_theCursor := dbms_sql.open_cursor;
 21
```

Начнем вложенный блок с обработчиком исключительных ситуаций. Если в этом блоке кода возникнет ошибка, необходимо обработать ее как можно ближе к месту возникновения и закрыть курсор, чтобы избежать "утечки курсоров", когда дескриптор открытого курсора просто теряется при распространении ошибки за пределы подпрограммы.

```
 22      begin
 23          -- Шаг 2, проанализировать запрос.
 24          dbms_sql.parse(c             => l_theCursor,
 25                         statement     => l_update,
 26                         language_flag => dbms_sql.native);
 27
 28          -- Шаг 3, связать все входные и выходные переменные.
 29          dbms_sql.bind_variable(c     => l_theCursor,
 30                                 name  => ':bv1',
 31                                 value => p_newDname);
 32          dbms_sql.bind_variable(c     => l_theCursor,
 33                                 name  => ':bv2',
 34                                 value => p_newLoc);
 35          dbms_sql.bind_variable(c     => l_theCursor,
 36                                 name  => ':pk',
```

```
                                value => p_deptno);
        dbms_sql.bind_variable(c     => l_theCursor,
                               name  => ':out',
                               value => p_rowid,
                               out_value_size => 4000);

```

Учтите, что, хотя **возвращаемые** переменные передаются как параметры в режиме **OUT,** необходимо связать их перед выполнением. Необходимо также указать наибольший размер ожидаемого результата (OUT_VALUE_SIZE), чтобы сервер Oracle выделил под него соответствующее пространство.

```
        -- Шаг 4: выполнить оператор. Поскольку это оператор ЯМД,
        -- в переменной L_STATUS окажется количество измененных строк.
        -- Именно это значение мы и возвращаем.

       l_status := dbms_sql.execute(l_theCursor);

        -- Шаг 5: выбрать OUT-переменные из результатов выполнения.
        dbms_sql.variable_value(c     => l_theCursor,
                                name  => ':out',
                                value => p_rowid);

        -- Шаг 6: закрыть курсор.
        dbms_sql.close_cursor(c => l_theCursor);
        return l_columnValue;
    exception
        when dup_val_on_index then
           dbms_output.put_line('==> ' || sqlerrm);
            dbms_sql.close_cursor(c => l_theCursor);
            RAISE;
   end;
end;
/
```

```
Function created.
```

```
scott@TKYTE816> set serveroutput on
scott@TKYTE816> declare
  2    l_rowid     varchar(50);
  3      l_rows    number;
  4  begin
  5      l_rows := update_row('SCOTT', 'CONSULTING', 'WASHINGTON',
                                      '10', l_rowid);
  6
  7     dbms_output.put_line('Updated ' || l_rows || ' rows');
  8     dbms_output.put_line('its rowid was ' || l_rowid);
  9  end;
 10  /
Updated 1 rows
its rowid was AAAGnuAAFAAAAESAAA

PL/SQL procedure successfully completed.
```

Итак, я продемонстрировал особенности использования пакета **DBMS_SQL** для выполнения блока кода с передачей входных данных и выборкой результатов. Повторю еще раз: представленный выше блок кода лучше реализовать с помощью встроенного динамического SQL (чуть ниже мы так и сделаем). Подпрограммы пакета **DBMS_SQL** в нем применялись в основном для демонстрации использования соответствующего функционального интерфейса. В других главах книги, в частности в главе 9, посвященной загрузке данных, продемонстрировано, почему пакет **DBMS_SQL** по-прежнему широко используется. В главе 9 рассмотрен код программ загрузки и выгрузки данных на PL/SQL. В них средства **DBMS_SQL** используются в полном объеме, позволяя обрабатывать неизвестное количество столбцов любого типа как во входных данных (для операторов **INSERT),** так и в результатах (для операторов **SELECT**).

Мы рассмотрели примерно 75 процентов функциональных возможностей пакета **DBMS_SQL.** Чуть позже, многократно выполняя один и тот же динамически сформированный оператор, мы рассмотрим взаимодействие с помощью массивов и сравним использование пакета **DBMS_SQL** и встроенного динамического SQL. Пока, однако, мы завершим обзор пакета **DBMS_SQL.** Полный список входящих в него подпрограмм и описание их входных/выходных параметров можно найти в руководстве *Oracle8i Supplied PL/SQL Packages Reference,* где отдельно рассмотрена каждая подпрограмма.

Встроенный динамический SQL

Встроенный динамический SQL впервые появился в Oracle 8i. Он позволяет декларативно выполнять динамический SQL в среде PL/SQL. Большинство действий можно выполнить с помощью одного оператора, EXECUTE IMMEDIATE, а остальные — с помощью оператора OPEN FOR. Оператор EXECUTE IMMEDIATE имеет следующий синтаксис:

```
EXECUTE IMMEDIATE 'оператор'
[INTO {переменная1., переменная2, ... переменнаяN | запись}]
[USING [IN | OOT | IN OUT] связываемая_переменная1, ...
связываемая_переменнаяN]
[{RETURNING | RETURN} INTO результат1 [, ..., результатN]...];
```

где:

- **оператор** — любой оператор SQL или PL/SQL-блок;
- **переменная1, переменная2,... переменнаяN** или **запись** — переменные **PL/SQL,** в которые необходимо выбрать данные (столбцы одной строки результатов оператора **SELECT);**
- **связываемая_переменная1,... связываемая переменнаяN** — набор переменных PL/SQL, используемых для передачи входных данных/результатов;
- **результат1,** ... **результатN** — набор PL/SQL-переменных, используемых для размещения результатов, возвращаемых конструкцией **RETURN** оператора ЯМД.

В следующем примере код для функций **GET_ROW_CNTS** и **UPDATE_ROW,** которые мы ранее реализовали с помощью средств пакета **DBMS_SQL,** переписан с использованием оператора **EXECUTE IMMEDIATE.** Начнем с функции **GET_ROW_CNTS:**

```
scott@TKYTE816> create or replace
  2  function get_row_cnts(p_tname in varchar2)  return number
  3  as
  4     l_cnt number;
  5  begin
  6          execute immediate
  7            'select count(*)  from ' || p_tname
  8             into l_cnt;
  9
 10          return l_cnt;
 11  end;
 12  /

Function created.

scott@TKYTE816> set serveroutput on
scott@TKYTE816> exec dbms_output.put_line(get_row_cnts('emp'));
14

PL/SQL procedure successfully completed.
```

Использовав оператор **SELECT...INTO...** в качестве аргумента для **EXECUTE IMMEDIATE,** мы существенно уменьшили объем кода. Девять процедурных шагов, необходимых при использовании пакета **DBMS_SQL,** превратились в один шаг в случае встроенного динамического SQL. Не всегда удается свести все к одному шагу — иногда необходимо три, как будет показано ниже, — но общая идея понятна. Встроенный динамический **SQL** в данном случае обеспечивает более высокую производительность при написании кода (последствия его использования с точки зрения производительности мы рассмотрим чуть позже). Также бросается в глаза отсутствие раздела **EXCEPTION** - обработка исключительных ситуаций не нужна, поскольку все происходит **неявно.** Нет курсора, который необходимо закрывать, ничего не нужно освобождать. Сервер Oracle все делает сам.

Теперь реализуем с помощью встроенного динамического SQL функцию **UPDATE_ROW:**

```
scott@TKYTE816> create or replace
  2  function update_row(p_owner    in varchar2,
  3                      p_newDname in varchar2,
  4                      p_newLoc   in varchar2,
  5                      p_deptno   in varchar2,
  6                      p_rowid    out varchar2)
  7  return number
  6  is
  9  begin
 10      execute immediate
 11                'update '  || p_owner ||  '.dept
 12                     set dname = :bv1, loc = :bv2
 13                   where deptno = to_number(:pk)
 14                returning rowid into :out'
 15      using p_newDname, p_newLoc, p_deptno
 16       returning into p_rowid;
 17
```

```
 18     return sql%rowcount;
 19  end;
 20  /

Function created.

scott@TKYTE816> set serveroutput on
scott@TKYTE816> declare
  2     l_rowid  varchar(50);
  3     l_rows   number;
  4  begin
  5     l_rows := update_row('SCOTT', 'CONSULTING',
  6                          'WASHINGTON', '10', l_rowid);
  7
  8     dbms_output.put_line('Updated ' || l_rows || ' rows');
  9     dbms_output.put_line('its rowid was ' || l_rowid);
 10  end;
 11  /
Updated 1 rows
its rowid was AAAGnuAAFAAAAESAAA

PL/SQL procedure successfully completed.
```

Снова код существенно сократился — один шаг вместо шести; такой код проще читать и сопровождать. В этих двух случаях встроенный динамический SQL, несомненно, превосходит средства пакета **DBMS_SQL.**

Помимо оператора **EXECUTE IMMEDIATE** встроенный динамический SQL поддерживает динамическую обработку *курсорных переменных,* **REF CURSOR**. Курсорные переменные достаточно давно поддерживаются сервером Oracle (с версии 7.2). Первоначально они позволяли открыть **(OPEN)** запрос (результирующее множество) в хранимой процедуре и передать ссылку на него клиенту. С их помощью хранимые процедуры возвращают результирующие множества клиентам при использовании VB, протоколов JDBC и ODBC или библиотеки **OCI**. Позднее, в версии 7.3, поддержка курсорных переменных была расширена, так что в PL/SQL появилась возможность использовать их не только в операторе **OPEN,** но и в операторе **FETCH** (в качестве клиента могла использоваться другая подпрограмма на PL/SQL). Это позволило подпрограмме на PL/SQL принимать результирующее множество в качестве входных данных и обрабатывать его. Таким образом, впервые стало возможно централизовать общую обработку результатов запросов: одна подпрограмма может выбирать данные из нескольких различных запросов (результирующих множеств). До появления версии Oracle 8i, однако, курсорные переменные по сути были исключительно статические. На этапе компиляции (при создании хранимой процедуры) надо было точно знать, какой SQL-запрос будет выполняться. Это было весьма существенное ограничение, поскольку не позволяло динамически изменять условия запроса, запрашивать другую таблицу и т.п. Начиная с Oracle 8i встроенный динамический SQL позволяет динамически открывать для запроса курсорную переменную. При этом используется следующий синтаксис:

```
OPEN курсорная_переменная FOR 'select ...'
USING связываемая_переменная1, связываемая_переменная2, ...;
```

Итак, с помощью курсорных переменных и динамического SQL можно реализовать обобщенную процедуру запроса таблицы в зависимости от входных данных и возвращения результирующего множества клиенту для дальнейшей обработки:

```
scott@TKYTE816> create or replace package my_pkg
  2  as
  3      type refcursor_Type is ref cursor;
  4
  5      procedure get_emps(p_ename in varchar2 default NULL,
  6                         p_deptno in varchar2 default NULL,
  7                         p_cursor in out refcursor_type);
  8  end;
  9  /

Package created.

scott@TKYTE816> create or replace package body my_pkg
  2  as
  3      procedure get_emps(p_ename in varchar2 default NULL,
  4                         p_deptno in varchar2 default NULL,
  5                         p_cursor in out refcursor_type)
  6      is
  7          l_query long;
  8          l_bind  varchar2(30);
  9      begin
 10         l_query := 'select deptno, ename, job from emp';
 11
 12          if (p_ename is not NULL)
 13          then
 14             l_query := l_query || ' where ename like :x';
 15             l_bind := '%' || upper(p_ename) || '%';
 16          elsif (p_deptno is not NULL)
 17          then
 18             l_query := l_query || ' where deptno = to_number(:x)';
 19              l_bind := p_deptno;
 20          else
 21              raise_application_error(-20001,'Missing search criteria');
 22          end if;
 23
 24          open p_cursor for l_query using l_bind;
 25      end;
 26  end;
 27  /

Package body created.

scott@TKYTE816> variable C refcursor
scott@TKYTE816> set autoprint on
scott@TKYTE816> exec my_pkg.get_emps(p_ename => 'a', p_cursor => :C)

PL/SQL procedure successfully completed.
```

```
DEPTNO ENAME      JOB

    20 ADAMS      CLERK
    30 ALLEN      SALESMAN
    30 BLAKE      MANAGER
    10 CLARK      MANAGER
    30 JAMES      CLERK
    30 MARTIN     SALESMAN
    30 WARD       SALESMAN

7 rows selected.

scott@TKYTE816> exec my_pkg.get_emps(p_deptno=> '10', p_cursor => :C)

PL/SQL procedure successfully completed.

DEPTNO ENAME      JOB

    10 CLARK      MANAGER
    10 KING       PRESIDENT
    10 MILLER     CLERK
```

Если созданный динамически запрос возвращает более одной строки, надо использовать представленный выше метод, а не оператор **EXECUTE IMMEDIATE**.

Итак, по сравнению с представленными выше подпрограммами пакета **DBMS_SQL**, использование операторов **EXECUTE IMMEDIATE** и **OPEN FOR** существенно упрощает написание программ. Значит ли это, что пакет **DBMS_SQL** больше использовать не придется? Определенно, — не значит. Представленные выше примеры показывают, насколько простым может быть использование динамического SQL, если количество связываемых переменных известно во время компиляции. Если бы мы этого не знали, то не смогли бы использовать оператор **EXECUTE IMMEDIATE** так просто, как в представленных примерах. Для этого оператора количество связываемых переменных надо знать заранее. Пакет **DBMS_SQL** в этом отношении обеспечивает большую гибкость. Помимо количества связываемых переменных необходимо знать еще и столбцы, входящие в результат выполнения SQL-оператора **SELECT**. Если количество и типы этих столбцов неизвестны, использовать оператор **EXECUTE IMMEDIATE** тоже не удастся. Можно будет использовать оператор **OPEN FOR**, если клиент, получающий курсорную переменную, не является другой подпрограммой на языке PL/SQL.

Оператор **EXECUTE IMMEDIATE** обеспечит более высокую производительность по сравнению с пакетом **DBMS_SQL** для всех операторов, анализируемых и выполняемых однократно (все наши примеры пока были именно такими). Для выполнения подпрограмм пакета **DBMS_SQL** в этом отношении требуется больше ресурсов, потому что нужно вызвать пять или шесть процедур там, где достаточно одного выполнения оператора **EXECUTE IMMEDIATE**.

Однако пакет **DBMS_SQL** обеспечивает более высокую производительность, если его процедуры используются для многократного выполнения одного и того же проанализированного оператора. Оператор **EXECUTE IMMEDIATE** не позволяет "повторно использовать" проанализированные операторы. Он всегда разбирает оператор, и расходы ресурсов на повторные выполнения этой операции со временем превышают расхо-

ды на дополнительные вызовы процедур. Особое значение это приобретает в многопользовательской среде. Наконец, операторы **EXECUTE IMMEDIATE** и **OPEN** не позволяют обрабатывать массивы так же просто, как подпрограммы пакета **DBMS_SQL** и, как будет продемонстрировано, одно это может принципиально повлиять на производительность.

Сравнение пакета DBMS_SQL и встроенного динамического SQL

Рассмотрев способы реализации подпрограмм с помощью пакета D B M S _ S Q L И встроенного динамического SQL, поговорим о том, когда следует использовать каждый из способов. Решение зависит от следующих факторов.

- Известно ли на этапе компиляции, какие связываемые переменные придется использовать? Если — нет, надо выбрать пакет **DBMS_SQL**.
- Известны ли на этапе компиляции все выходные данные? Если — нет, нужно отдать предпочтение пакету **DBMS_SQL**.
- Надо ли использовать курсорную переменную для возврата результирующего множества из хранимой процедуры? Если — да, придется использовать оператор **OPEN FOR.**
- Будет ли формируемый динамически оператор выполняться в сеансе один раз или многократно. Если один и тот же динамически формируемый оператор выполняется несколько раз, пакет **DBMS_SQL** обеспечит более высокую производительность.
- Надо ли использовать обработку массивов для динамически формируемых операторов.

Три из этих факторов мы рассмотрим ниже (на самом деле — даже четыре, поскольку многократное выполнение оператора мы рассмотрим как в случае обработки массивов, так и без оной).

Связываемые переменные

Связываемые переменные существенно влияют на производительность системы. Если они не используются, производительность недопустимо низка. Метод *автоматической подстановки связываемых переменных* (auto binding) путем установки соответствующего значения параметра **CURSOR_SHARING** был рассмотрен в главе 10. Это несколько улучшает ситуацию, но приводит к избыточному расходованию ресурсов, поскольку сервер вынужден переписывать запрос и удалять информацию, существенную для оптимизатора, которую можно было бы оставить при явном включении связываемых переменных в код.

Предположим, необходимо создать процедуру, динамически создающую запрос на основе введенных пользователем данных. Запрос всегда будет возвращать однотипные результаты (тот же список выбора), но конструкция **WHERE** будет меняться в зависимости от входных данных. Из соображений производительности необходимо использо-

вать связываемые переменные. Как это сделать с помощью встроенного динамического SQL и средств пакета DBMS_SQL? Чтобы представить методы, начну со спецификации подпрограммы. Разрабатываемая процедура будет иметь следующий вид:

```
scott@TKYTE816> create or replace package dyn_demo
  2 as
  3     type array is table of varchar2(2000);
  4
  5
  6     /*
  7      * DO_QUERY будет динамически запрашивать таблицу emp
  8      * и обрабатывать результаты. Ее можно вызвать
  9      * следующим образом:
 10      *
 11      * dyn_demo.do_query( array('ename', 'job'),
 12             array('like',                '='),
 13                          array('%A%', 'CLERK'));
 14      *
 15      * для выполнения запроса:
 16      *
 17      * select * from emp where ename like '%A%' and job = 'CLERK'
 18      *
 19      * например.
 20      */
 21     procedure do_query(p_cnames    in array,
 22                        p_operators in array,
 23                        p_values    in array);
 24
 25 end;
 26 /

Package created.
```

Вполне естественно реализовать ее с помощью DBMS_SQL — для таких ситуаций и создавался этот пакет. Можно организовать цикл по массивам столбцов и значений и построить конструкцию WHERE. Затем проанализировать запрос и выполнить еще один цикл по массивам для связывания значений переменных. После этого выполнить оператор, выбрать строки и обработать их. Это можно записать следующим образом:

```
scott@TKYTE816> create or replace package body dyn_demo
  2 as
  3
  4 /*
  5  * Реализация динамического запроса с неизвестными
  6  * связываемыми переменными средствами DBMS_SQL
  7  */
  8 g_cursor int default dbms_sql.open_cursor;
  9
 10
 11 procedure do_query(p_cnames     in array,
 12                     p_operators in array,
 13                     p_values    in array)
```

```
14  is
15      l_query      long;
16      l_sep      varchar2(20) default ' where ';;
17      l_comma     varchar2(1) default '';
18      l_status    int;
19      l_colValue  varchar2(4000);
20  begin
21      /*
22       * Это наш постоянный список выбора — мы всегда
23       * выбираем эти три столбца. Изменяются
24       * условия выбора.
25       */
26      l_query := 'select ename, empno, job from emp';
27
28      /*
29       * Мы строим условие, сначала
30       * помещая в запрос конструкцию:
31       *
32       * ename operator :bvX
33       *
34       */
35      for i in 1 .. p_cnames.count loop
36          l_query := l_query || l_sep || p_cnames(i) || ' ' ||
37                                         p_operators(i) || ' ' ||
38                                         ':bv' || i;
39          l_sep := ' and •;
40      end loop;
41
42      /*
43       * Теперь можно анализировать запрос
44       */
45      dbms_sql.parse(g_cursor, l_query, dbms_sql.native);
46
47      /*
48       * и определять столбцы результата. Все три столбца
49       * выбираются в переменные типа VARCHAR2.
50       */
51      for i in 1 .. 3 loop
52          dbms_sql.define_column(g_cursor, i, l_colValue, 4000);
53      end loop;
54
55      /*
56       * Теперь можно связать входные переменные запроса
57       */
58      for i in 1 .. p_cnames.count loop
59          dbms_sql.bind_variable(g_cursor, ':bv'||i, p_values(i), 4000);
60      end loop;
61
62      /*
63       * и выполнить его. Так формируется результирующее множество
64       */
```

```
65      l_status  := dbms_sql.execute(g_cursor);
66
67      /*
68       * теперь проходим в цикле по строкам и выдаем результаты.
69       */
70      while (dbms_sql.fetch_rows(g_cursor) > 0)
71      loop
72          l_comma := '';
73          for i in 1 .. 3 loop
74              dbms_sql.column_value(g_cursor, i, l_colValue);
75              dbms_output.put(l_comma || l_colValue);
76      l_comma     :=     ', ';
77          end loop;
78          dbms_output.new_line;
79      end loop;
80  end;
81
82  end dyn_demo;
83  /

Package body created.

scott@TKYTE816> set serveroutput on
scott@TKYTE816> begin
  2        dyn_demo.do_query(dyn_demo.array('ename',  'job'),
  3                          dyn_demo.array('like',  '='),
  4                          dyn_demo.array('%A%',  'CLERK'));
  5  end;
  6  /
ADAMS,7876,CLERK
JAMES,7900,CLERK

PL/SQL procedure successfully completed.
```

Как видите, все просто и в рамках действий, предусмотренных для использования пакета **DBMS_SQL**. Теперь реализуем то же самое с помощью встроенного динамического SQL Здесь мы сталкиваемся с проблемой. Для динамического выполнения запроса со связываемыми переменными во встроенном динамическом SQL используется следующий синтаксис:

OPEN курсорная_переменная FOR 'select____'
USING переиенная1, переменная2, переменная3, ...;

Проблема в том, что на этапе компиляции мы не знаем размера списка **USING** — будет ли в нем одна переменная, две или вообще ни одной? Поэтому необходимо параметризировать запрос, но использовать обычные связываемые переменные нельзя. Можно, однако, использовать средство, предназначавшееся совсем для других целей. В главе 21, при изучении средств тщательного контроля доступа, мы рассмотрим *контекст приложения* (application context) и его использование. Контекст приложения, по сути, позволяет поместить в *пространство имен* (namespace) пару переменная/значение. К этой паре переменная/значение можно обращаться в SQL-операторах с помощью встроенной функции **SYS_CONTEXT**. Контекст приложения, таким образом, можно исполь-

зовать для параметризации запроса, помещая связываемые значения в пространство имени и выбирая их в запросе с помощью встроенной функции SYS_CONTEXT.

Итак, вместо запроса следующего вида:

```
select ename, empno, job
  from emp
 where ename like :bvl
   and job = :bv2;
```

создаем такой запрос:

```
select ename, empno, job
  from emp
 where ename like SYS_CONTEXT('namespace','ename')
   and job = SYS_CONTEXT('namespace','job');
```

Код, реализующий этот метод, может выглядеть так:

```
scott@TKYTE816> REM Пользователь SCOTT должен иметь привилегию CREATE ANY
->CONTEXT
scott@TKYTE816> REM или роль с такой привилегий, иначе код не сработает
scott@TKYTE816> create or replace context bv_context using dyn_demo
  2  /

Context created.

scott@TKYTE816> create or replace package body dyn_demo
  2  as
  3
  4  procedure do_query(p_cnames    in array,
  5                     p_operators in array,
  6                     p_values    in array)
  7  is
  8      type rc is ref cursor;
  9
 10      l_query      long;
 11      l_sep        varchar2(20) default ' where ';
 12      l_cursor     rc;
 13      l_ename      emp.ename%type;
 14      l_empno      emp.empno%type;
 15      l_job        emp.job%type;
 16  begin
 17      /*
 18       * Это наш постоянный список выбора — мы всегда
 19       * выбираем эти три столбца. Изменяются
 20       * условия выбора.
 21       */
 22      l_query := 'select ename, empno, job from emp';
 23
 24      for i in 1 .. p_cnames.count loop
 25         l_query := l_query || l_sep | |
 26                    p_cnames(i) || ' ' ||
 27                    p_operators(i) || ' ' ||
 28            'sys_context(''BV_CONTEXT'',''' ||
 29                                  p_cnames(i)  ||  ''')';
```

```
 30         l_sep := ' and ';
 31          dbms_session.set_context('bv_context',
 32                                   p_cnames(i),
 33                                   p_values(i));
 34      end loop;
 35
 36      open l_cursor for l_query;
 37      loop
 38          fetch l_cursor into l_ename, l_empno, l_job;
 39          exit when l_cursor%notfound;
 40         dbms_output.put_line(l_ename M ', ' ||l_empno ||',' || l_job);
 41      end loop;
 42      close l_cursor;
 43 end;
 44
 45  end dyn_demo;
 46 /

Package body created.

scott@TKYTE816> set serveroutput on
scott@TKYTE816> begin
  2       dyn_demo.do_query( dyn_demo.array('ename',  'job'),
  3                          dyn_demo.array('like',  '='),
  4               dyn_demo.arrayC%A%',                'CLERK'));
  5  end;
  6  /
ADAMS,7876,CLERK
JAMES,7900,CLERK

PL/SQL procedure successfully completed.
```

Так что, с точки зрения использования связываемых переменных, все гораздо сложнее, чем в случае пакета **DBMS_SQL,** — необходим **хитрый прием.** После того, как вы поймете суть этого приема, вполне можно использовать встроенный динамический SQL вместо средств пакета **DBMS_SQL, если только запрос выдает фиксированное количество результатов и используется контекст приложения.** Чтобы эффективно решать с помощью встроенного динамического SQL подобного рода задачи, необходимо создать и использовать контекст приложения. В конечном итоге оказывается, что представленный выше пример с курсорными переменными при реализации с помощью встроенного динамического SQL работает быстрее. В случае простых запросов, когда временем обработки самого запроса можно пренебречь, встроенный динамический SQL обеспечивает скорость выборки данных почти вдвое выше, чем пакет **DBMS_SQL.**

Количество столбцов выходных данных на этапе компиляции не известно

Здесь все понятно: если клиент, выбирающий и обрабатывающий данные, создается на PL/SQL, необходимо использовать пакет **DBMS_SQL.** Если клиент, выбирающий и обрабатывающий данные, — приложение на процедурном языке программирования, ис-

пользующее интерфейсы ODBC, JDBC, OCI и т.п., необходимо использовать встроенный динамический SQL.

Рассмотрим ситуацию, когда, получая запрос во время выполнения, мы не знаем, сколько столбцов входит в список выбора. Необходимо определить это в коде PL/SQL. Оказывается, встроенный динамический SQL использовать нельзя, поскольку придется включить в код оператор вида:

```
FETCH курсор INTO переменная!, переменная2, переменная3, ...;
```

но сделать этого нельзя, потому что до момента выполнения не известно, сколько переменных надо в него поместить. Это один из случаев, когда придется использовать средства пакета **DBMS_SQL**, поскольку он позволяет применять следующие конструкции:

```
41      while (dbms_sql.fetch_rows(l_theCursor) > 0)
42      loop
43          /* Строим длинную строку результатов, — это эффективнее, чей
44           * вызывать DBMS_OUTPUT.PUT_LINE в цикле.
45           */
46          l_cnt := l_cnt+1;
47          l_line := l_cnt;
48          /* Шаг 8 — получить и обработать данные столбцов. */
49          for i in 1 .. l_colCnt loop
50              dbms_sql.column_value(l_theCursor, i, l_columnValue);
51              l_line := l_line || ',' || l_columnValue;
52          end loop;
53
54          /* Теперь выдаем строку. */
55          dbms_output.put_line(l_line);
56      end loop;
```

Можно проходить по столбцам в цикле, как если бы они представляли собой массив. Представленная выше конструкция взята из следующего фрагмента кода:

```
scott@TKYTE816> create or replace
  2  procedure dump_query(p_query in varchar2)
  3  is
  4      l_columnValue   varchar2(4000);
  5      l_status        integer;
  6      l_colCnt        number default 0;
  7      l_cnt           number default 0;
  8      l_line          long;
  9
 10      /* Мы будем использовать эту таблицу, чтобы узнать,
 11       * сколько столбцов придется выбирать, чтобы определить их,
 12       * а затем выбрать их значения.
 13       */
 14      l_descTbl       dbms_sql.desc_tab;
 15
 16
 17     /* Шаг 1: открыть курсор. */
 18      l_theCursor     integer default dbms_sql.open_cursor;
 19  begin
 20
```

```
 21      /* Шаг 2: проанализировать запрос, чтобы можно было получить
-> описание его результатов. */
 22      dbms_sql.parse(l_theCursor, p_query, dbms_sql.native);
 23
 24      /* Шаг 3: получаем описание результатов запроса. */
 25    dbms_sql.describe_columns(l_theCursor,    l_colCnt, l_descTbl);
 26
 27      /* Шаг 4 в этом примере не используется, потому что связывать
-> ничего не нужно.
 28       * Шаг 5: необходимо определить каждый столбец, сообщить серверу,
 29       * что и куда мы будем выбирать. В данном случае все данные
 30       * будут выбираться в одну локальную переменную типа
-> varchar2(4000).
 31       */
 32      for i in 1 .. l_colCnt
 33      loop
 34         dbms_sql.define_column(l_theCursor, i, l_columnValue, 4000);
 35       end loop;
 36
 37      /* Шаг 6: выполнить оператор. */
 38      l_status := dbms_sql.execute(l_theCursor);
 39
 40      /* Шаг 7: выбрать все строки. */
 41      while (dbms_sql.fetch_rows(l_theCursor) > 0)
 42      loop
 43          /* Строим длинную строку результатов — это эффективнее, чем
 44           * вызывать DBMS_OUTPOT.PUT_LINE в цикле.
 45           */
 46          l_cnt :- l_cnt+l;
 47          l_line := l_cnt;
 48          /* Шаг 8: получаем и обрабатываем данные столбцов. */
 49          for i in 1 .. l_colCnt loop
 50              dbms_sql.column_value(l_theCursor, i, l_columnValue);
 51              l_line := l_line || ',' || l_columnValue;
 52          end loop;
 53
 54          /* Теперь выдаем строку. */
 55          dbms_output.put_line(l_line);
 56      end loop;
 57
 58      /* Step 9: закрываем курсор, чтобы освободить ресурсы. */
 59      dbms_sql.close_cursor(l_theCursor);
 60  exception
 61      when others then
 62          dbms_sql.close_cursor(l_theCursor);
 63          raise;
 64  end dump_query;
 65  /

Procedure created.
```

Из этого следует, что пакет **DBMS_SQL** позволяет с помощью процедуры **DBMS_SQL.DESCRIBE_COLUMNS** получить количество, имена и типы данных стол-

бцов в запросе. В качестве примера ее использования рассмотрим обобщенную процедуру сброса результатов запроса в файл операционной системы. Она отличается от SQL-Unloader, рассмотренной в главе 9 при создании средств выгрузки данных. В данном примере данные сбрасываются в файл в виде записей фиксированной длины, в которых столбцы всегда начинаются с одной и той же позиции. Для этого анализируются результаты вызова **DBMS_SQL.DESCRIBE_COLUMNS,** в которых помимо количества выбираемых столбцов можно найти и их максимальный размер. Прежде чем рассмотреть пример полностью, давайте подробней разберемся с процедурой **DESCRIBE_COLUMNS.** После анализа запроса с помощью этой процедуры можно обратиться к серверу за информацией о том, что можно ожидать при выборке данных по этому запросу. Эта процедура создает массив записей с информацией об именах, типах данных, максимальном размере столбцов и т.п.

Вот пример использования процедуры **DESCRIBE_COLUMNS**. Выдаются данные, возвращаемые ею для запроса; благодаря этому можно узнать, какая информация доступна:

```
scott@TKYTE816> create or replace
  2  procedure  desc_query(p_query in varchar2)
  3  is
  4       l_columnValue   varchar2(4000);
  5       l_status        integer;
  6       l_colCnt        number default 0;
  7       l_cnt           number default 0;
  8       l_line          long;
  9
 10      /* На используем эту таблицу, чтобы узнать, какие данные
w  выбирает запрос
 11       */
 12      l_descTbl       dbms_sql.desc_tab;
 13
 14
 15     /* Шаг 1: открыть курсор. */
 16      l_theCursor     integer default dbms_sql.open_cursor;
 17  begin
 18
 19      /* Шаг 2: проанализировать входной запрос, чтобы можно было
w описать его результаты.  */
 20     dbms_sql.parse(l_theCursor, p_query, dbms_sql.native);
 21
 22      /* Шаг 3: описываем результаты запроса.
 23       * Переменная L_COLCNT будет содержать количество выбранных в
 24       * запросе столбцов. Оно будет равно L_DESCTBL.COUNT;
 25       * эта переменная содержит избыточную информацию. Переменная
 26       * L_DESCTBL содержит полезные сведения о выбранных столбцах.
 27       */
 28
 29      dbms_sql.describe_columns(c       => l_theCursor,
 30                                col_cnt => l_colCnt,
 31                                desc_t  => l_descTbl);
 32
```

```
33       for i in 1 .. l_colCnt
34      loop
35          dbms_output.put_line
36           ('Column Type.....' || l_descTbl(i).col_type);
37          dbms_output.put_line
38           ('Max Length.....' || l_descTbl(i).col_max_len);
39          dbms_output.put_line
40          ('Name.....' || l_descTbl(i).colname);
41          dbms_output.put_line
42           ('Name Length.....' || l_descTbl(i) .col_name_len);
43          dbms_output.put_line
44          ('ObjColumn Schema Name.' || l_descTbl(i).col_schema_name);
45          dbms_output.put_line
46          ('Schema Name Length....' || l_descTbl(i) .col_schema_name_len) ;
47          dbms_output.put_line
48          ('Precision.....' || l_descTbl(i) ,col_precision) ;
49          dbms_output.put_line
50          ('Scale.....' || l_descTbl(i).col_scale);
51          dbms_output.put_line
52          ('Charsetid.....' || l_descTbl (i). col_Charsetid) ;
53          dbms_output.put_line
54           ('Charset Form.....' || l_descTbl(i) .col_charsetform) ;
55          if (l_desctbl(i).col_null_ok) then
56              dbms_output.put_line( 'Nullable.....Y');
57          else
58              dhms_output.put_line( 'Nullable.....N') ;
59          end if;
60             dbms_output.put_line('-------------------------') ;
61       end loop;
62
63      /* Шаг 9: закрыть курсор и освободить ресурсы. */
64      dbms_sql.close_cursor(l_theCursor);
65  exception
66      when others then
67          dbms_sql.close_cursor(l_theCursor);
68          raise;
69  end desc_query;
70  /

Procedure created.

scott@TKYTE816> set serveroutput on
scott@TKYTE816> exec desc_query('select rowid, ename from emp');
Column Type.....11
Max Length.....16
Name.....ROWID
Name Length.....5
ObjColumn Schema Name.
Schema Name Length.... 0
Precision.....0
Scale.....0
Charsetid.....0
```

```
Charset Form.........0
Nullable.............Y

Column Type..........1
Max Length...........10
Name.................ENAME
Name Length..........5
ObjColumn Schema Name.
Schema Name Length....0
Precision............0
Scale................0
Charsetid............31
Charset Form.........1
Nullable.............Y

PL/SQL procedure successfully completed.
```

К сожалению, значение **COLUMN TYPE** — число, а не имя типа данных, так что если не знать, что значение 11 соответствует типу **ROWID,** а значение 1 — типу **VARCHAR2,** расшифровать эти результаты не удастся. В руководстве *Oracle Call Interface Programmer's Guide* представлен полный список внутренних кодов типов данных и соответствующих имен типов. Этот список воспроизведен ниже.

VARCHAR2, NVARCHAR2	1
NUMBER	2
LONG	8
ROWID	11
DATE	12
RAW	23
LONG RAW	24
CHAR, NCHAR	96
Пользовательский тип (объектный тип, VARRAY, вложенная таблица)	108
REF	111
CLOB, NCLOB	112
BLOB	113
BFILE	114
UROWID	208

Теперь мы готовы рассмотреть всю подпрограмму, которая может принять практически любой запрос и сбросить результаты его выполнения в файл операционной системы (предполагается, что пакет **UTL_FILE** настроен; эта настройка подробно описана в приложении А):

```
scott@TKYTE816> create or replace
  2  function  dump_fixed_width(p_query     in varchar2,
  3                             p_dir       in varchar2,
  4                             p_filename  in varchar2)
  5  return number
  6  is
  7      l_output        utl_file.file_type;
  8      l_theCursor     integer default dbms_sql.open_cursor;
  9      l_columnValue   varchar2(4000);
 10      l_status        integer;
 11      l_colCnt        number default 0;
 12      l_cnt           number default 0;
 13       l_line          long;
 14      l_descTbl       dbms_sql.desc_tab;
 15      l_dateformat    nls_session_parameters.value%type;
 16  begin
 17      select value into l_dateformat
 18        from nls_session_parameters
 19       where parameter = 'NLS_DATE_FORMAT';
 20
 21      /* Используем формат даты, включающий время. */
 22      execute immediate
 23       'alter session set nls_date_format=''dd-mon-yyyy hh24:mi:ss'' ';
 24     l_output := utl_file.fopen(p_dir, p_filename, 'w', 32000);
 25
 26      /* Анализируем входной запрос, чтобы можно было получить его
->  описание. */
 27     dbms_sql.parse(l_theCursor,  p_query, dbms_sql.native);
 28
 29      /* Теперь получаем описание результатов запроса. */
 30     dbms_sql.describe_columns(l_theCursor, l_colCnt, l_descTbl);
 31
 32      /* Необходимо определить каждый столбец и указать серверу,
 33       * что и куда мы будем выбирать. В данном случае, все данные
 34       * будут выбираться в одну переменную типа varchar2(4000).
 35       *
 36       * Мы также определим максимальный размер каждого столбца. Это
 37       * делается для того, чтобы при выдаче данных каждое поле
 38       * начиналось и заканчивалось в одной и той же позиции в каждой
->  записи.
 39       */
 40      for i in 1 .. l_colCnt loop
 41         dbms_sql.define_column(l_theCursor, i, l_columnValue, 4000);
 42
 43          if (l_descTbl(i).col_type = 2)   /* тип number */
 44          then
 45              L_descTbl(i).col_max_len := l_descTbl(i) .col_precision+2;
 46          elsif (l_descTbl(i).col_type = 12)  /* тип date */
 47          then
 48               /* длина заданного выше формата даты */
 49        l_descTbl(i).col_max_len        := 20;
```

```
 50         end if;
 51       end loop;
 52
 53      l_status  := dbms_sql.execute(l_theCursor);
 54
 55       while (dbms_sql.fetch_rows(l_theCursor) > 0)
 56       loop
 57          /* Строим большую строку результата. Это более эффективно,
 58           * чем вызывать процедуру UTL_FILE.PUT в цикле.
 59           */
 60          l_line := null;
 61          for i in 1 .. l_colCnt loop
 62              dbms_sql.column_value(l_theCursor, i, l_columnValue);
 63              l_line := l_line ||
 64            rpad(nvl(l_columnValue,           ' ' ) ,
 65                         l_descTbl(i).col_max_len);
 66          end loop;
 67
 68           /* Теперь выдаем строку в файл и увеличиваем значение
->  счетчика. */
 69          utl_file.put_line(l_output, l_line);
 70          l_cnt := l_cnt+1;
 71      end loop;
 72
 73      /* Освобождаем ресурсы. */
 74      dbms_sql.close_cursor(l_theCursor);
 75      utl_file.fclose(l_output);
 76
 77      /* Восстанавливаем формат даты ... и завершаем работу. */
 78      execute immediate
 79      'alter session set nls_date_format=''' || l_dateformat || ''' ';
 80      return l_cnt;
 81  exception
 82      when others then
 83          dbms_sql.close_cursor(l_theCursor) ;
 84          execute immediate
 85         'alter session set nls_date_format=' '' t || l_dateformat || ''' ';
 86
 87  end dump_fixed_width;
 88  /

Function created.
```

Итак, эта функция использует подпрограмму **DBMS_SQL.DESCRIBE_COLUMNS** для поиска количества столбцов и их типов данных. Я изменил некоторые значения максимальных размеров, чтобы учесть используемый формат даты, а также десятичную запятую и знак в числах. Представленная выше подпрограмма не может выгрузить данные типа **LONG**, **LONG RAW**, **CLOB** и **BLOB**. Ее легко изменить для поддержки данных типа **CLOB** и даже **LONG**. Придется специальным образом выполнять связывание переменных этих типов, а также использовать пакет DBMS_CLOB для выборки данных типа **CLOB** и подпрограмму **DBMS_SQL.COLUMN_VALUE_LONG** — для данных типа

LONG. Следует заметить, что добиться этого с помощью встроенного динамического SQL **невозможно** — его нельзя использовать, если список выбора в PL/SQL не известен.

Многократное выполнение одного и того же оператора

В данном случае придется выбирать между средствами пакета **DBMS_SQL** и встроенным динамическим SQL. За счет большего объема и сложности кода можно достичь более высокой производительности. Чтобы продемонстрировать это, я создам подпрограмму, динамически вставляющую в таблицу большое количество строк. В ней используется динамический SQL, поскольку до начала выполнения имя таблицы, куда будут вставляться данные, неизвестно. Для сравнения создадим четыре аналогичных подпрограммы:

Подпрограмма	*Назначение*
DBMSSQL_ARRAY	Использует обработку массивов в PL/SQL для множественной вставки строк
NATIVE_DYNAMIC_ARRAY	Использует эмуляцию обработки массивов с помощью таблиц объектного типа
DBMSSQL_NOARRAY	Выполняет построчную обработку при вставке строк
NAT1VE_DYNAMIC_NOARRAY	Выполняет построчную обработку при вставке строк

Первый метод (используемый в подпрограмме **DBMSSQL_ARRAY)** наиболее масштабируем и обеспечивает наибольшую производительность. В моих тестах на различных платформах первый и второй методы были очень **близки** по результатам в однопользовательской среде: если на машине не работают другие пользователи, они более-менее сопоставимы. На некоторых платформах встроенный динамический SQL работал немного быстрее, на других — пакет **DBMS_SQL. В** многопользовательской среде, однако, из-за повторного полного анализа запроса при каждом выполнении во встроенном динамическом SQL, подход с использованием обработки массивов средствами пакета **DBMS_SQL** обеспечивал лучшую масштабируемость. При этом не нужно было выполнять мягкий разбор запроса при каждом выполнении. Необходимо также учесть, что для эмуляции обработки массивов во встроенном динамическом SQL пришлось применить трюк. Так что код в обоих случаях оказался достаточно сложным. Обычно код, где используется встроенный динамический SQL, намного проще, чем код с вызовами **DBMS_SQL,** но не в этом случае.

Единственный определенный вывод, который можно сделать, — третий и четвертый методы намного медленнее первых двух. Следующие результаты были получены на платформе Solaris для одного пользователя, но результаты на платформе Windows были аналогичными. Выполните тесты на своей платформе, чтобы получить наиболее достоверные результаты.

```
scott@TKYTE816> create or replace type vcArray as table of varchar2(400)
  2  /

Type created.

scott@TKYTE816> create or replace type dtArray as table of date
  2  /
```

```
Type created.

scott@TKYTE816> create or replace type nmArray as table of number
  2  /

Type created.
```

Эти типы необходимы для эмуляции обработки массивов с помощью встроенного динамического SQL. Массивы именно этих типов мы и будем использовать (во встроенном динамическом SQL вообще нельзя использовать PL/SQL-таблицы). Теперь представим спецификацию пакета, который будет использоваться для тестов:

```
scott@TKYTE816> create or replace package load_data
  2  as
  3
  4  procedure dbmssql_array(p_tname     in varchar2,
  5                          p_arraysize in number default 100,
  6                          p_rows      in number default 500);
  7
  8  procedure dbmssql_noarray(p_tname  in varchar2,
  9                            p_rows   in number default 500);
 10
 11
 12  procedure native_dynamic_noarray(p_tname  in varchar2,
 13                                   p_rows   in number default 500);
 14
 15  procedure native_dynamic_array(p_tname     in varchar2,
 16                                 p_arraysize in number default 100,
 17                                 p_rows      in number default 500);
 18  end load_data;
 19  /

Package created.
```

Каждая из представленных выше процедур будет динамически вставлять строки в таблицу, заданную параметром P_TNAME. Количество вставляемых строк определяет параметр P_ROWS; при использовании обработки массивов их размер задается параметром P_ARRAYSIZE. Теперь переходим к реализации:

```
scott@TKYTE816> create or replace package body load_data
  2  as
  3
  4  procedure dbmssql_array(p_tname     in varchar2,
  5                          p_arraysize in number default 100,
  6                          p_rows      in number default 500)
  7  is
  8      l_stmt      long;
  9      l_theCursor integer;
 10      l_status    number;
 11      l_col1      dbms_sql.number_table;
 12      l_col2      dbms_sql.date_table;
 13      l_col3      dbms_sql.varchar2_table;
 14      l_cnt       number default 0;
 15  begin
```

```
16      l_stmt := 'insert into ' || p_tname ||
17              ' q1 (a, b, c) values (:a, :b, :c)';
18
19       l_theCursor := dbms_sql.open_cursor;
20      dbms_sql.parse(l_theCursor, l_stmt, dbms_sql.native);
21          /*
22           * Здесь мы будем формировать данные. После формирования
23           * ARRAYSIZE строк, мы вставляем их все сразу. В конце
24           * цикла, если еще остались строки, мы их тоже вставляем.
25           */
26      for i in 1 .. p_rows
27       loop
28             l_cnt := l_cnt+1;
29          l_coll(l_cnt) := i;
30      l_col2(l_cnt)      := sysdate+i;
31      l_col3(l_cnt)      := to_char(i);
32
33          if (l_cnt = p_arraysize)
34          then
35            dbms_sql.bind_array(l_theCursor, ':a', l_coll, 1, l_cnt);
36            dbms_sql.bind_array(l_theCursor, ':b', l_col2, 1, l_cnt);
37            dbms_sql.bind_array(l_theCursor, ':c', l_col3, 1, l_cnt);
38             l_status := dbms_sql.execute(l_theCursor);
39            l_cnt := 0;
40         end if;
41      end loop;
42      if (l_cnt > 0)
43      then
44           dbms_sql.bind_array(l_theCursor, ':a', l_coll, 1, l_cnt);
45           dbms_sql.bind_array(l_theCursor, ':b', l_col2, 1, l_cnt);
46           dbms_sql.bind_array(l_theCursor, ':c', l_col3, 1, l_cnt);
47            l_status := dbms_sql.execute(l_theCursor);
48     end if;
49      dbms_sql.close_cursor(l_theCursor);
50 end;
51
```

Итак, в этой подпрограмме используются средства пакета **DBMS_SQL** для вставки массива из N строк с помощью одной операции. Мы используем перегруженную подпрограмму **BIND_VARIABLE,** позволяющую пересылать PL/SQL-таблицу соответствующего типа с загружаемыми данными. Мы также указываем границы массива, чтобы сервер Oracle "знал", где начинается и заканчивается блок данных в переданной PL/SQL-таблице. В данном случае всегда следует начинать с индекса 1 и заканчивать индексом **L_CNT.** Обратите внимание, что для имени таблицы в операторе **INSERT** задан псевдоним (корреляционное имя) **Q1**. Я сделал это для того, чтобы при анализе производительности с помощью утилиты **TKPROF можно** было определить, какие операторы **INSERT** использовались той или иной подпрограммой. Вообще, код получается довольно простым. Теперь представим реализацию на базе пакета **DBMS_SQL** без обработки массивов:

```
52 procedure dbmssql_noarray(p_tname     in varchar2,
53                           p_rows      in number default 500)
54 is
55     l_stmt      long;
56     l_theCursor integer;
57     l_status    number;
58 begin
59    l_stmt := 'insert into ' || p_tname ||
60              ' q2 (a, b, c) values (:a, :b, :c)';
61
62     l_theCursor := dbms_sql.open_cursor;
63    dbms_sql.parse(l_theCursor, l_stmt, dbms_sql.native);
64         /*
65          * Здесь мы будем формировать данные.
66          * Каждая строка вставляется отдельным оператором
67          * в цикле.
68          */
69    for i in 1 .. p_rows
70     loop
71        dbms_sql.bind_variable(l_theCursor, ':a', i);
72        dbms_sql.bind_variable(l_theCursor, ':b', sysdate+i);
73        dbms_sql.bind_variable(l_theCursor, ':c', to_char(i));
74         l_status := dbms_sql.execute(l_theCursor);
75     end loop;
76     dbms_sql.close_cursor(l_theCursor);
77 end;
78
```

Эта подпрограмма отличается от предыдущей только тем, что не формируются массивы. Если вы пишете код, подобный этому, советую прибегнуть к обработке массивов. Как вскоре будет показано, это может существенно повысить производительность приложения. Теперь переходим к подпрограмме, использующей встроенный динамический SQL:

```
79 procedure native_dynamic_noarray(p_tname  in varchar2,
80                                   p_rows    in number default 500)
81 is
82 begin
83         /*
84          * Здесь мы формируем строку и вставляем ее.
85          * Что может быть проще для написания и выполнения!
86          */
87    for i in 1 .. p_rows
88     loop
89        execute immediate
90             'insert into ' || p_tname ||
91            ' q3 (a, b, c) values (:a, :b, :c)'
92       using i, sysdate+i, to_char(i);
93    end loop;
94 end;
95
```

В этой подпрограмме массивы не обрабатываются. Простенькая программа; ее легко создать, но вот производительность будет очень низкой из-за постоянно выполняемых разборов оператора. Наконец, пример эмуляции вставки массивов с помощью встроенного динамического SQL:

```
 96 procedure native_dynamic_array(p_tname     in varchar2,
 97                                p_arraysize in number default 100,
 98                                p_rows      in number default 500)
 99 is
100     l_stmt      long;
101     l_theCursor integer;
102     l_status    number;
103     l_coll      nmArray := nmArray();
104     l_col2      dtArray := dtArray();
105     l_col3      vcArray := vcArray();
106     l_cnt       number  := 0;
107 begin
108         /*
109          * Здесь мы будем формировать данные. После формирования
110          * ARRAYSIZE строк мы вставляем их все сразу. В конце цикла,
111          * если еще остались строки, мы их тоже вставляем.
112          */
113    l_coll.extend(p_arraysize);
114    l_col2.extend(p_arraysize);
115    l_col3.extend(p_arraysize);
116     for i in 1 .. p_rows
117     loop
118         l_cnt := l_cnt+1;
119         l_coll(l_cnt) := i;
120     l_col2(l_cnt)     := sysdate+i;
121     l_col3(l_cnt)     := to_char(i);
122
123         if (l_cnt = p_arraysize)
124         then
125                   execute immediate
126                    'begin
127                   forall i in 1 .. :n
128                           insert into ' || p_tname || '
129                  q4 (a, b, c) values (:a(i), :b(i), :c(i));
130                   end;'
131                   USING l_cnt, l_coll, l_col2, l_col3;
132                   l_cnt := 0;
133        end if;
134     end loop;
135     if (l_cnt > 0)
136     then
137           execute immediate
138           'begin
139          forall i in 1 .. :n
140                 insert into ' || p_tname || '
141                  q4 (a, b, c) values (:a(i), :b(i), :c(i));
```

```
142             end;'
143            USING l_cnt, l_col1, l_col2, l_col3;
144      end if;
145  end;
146
147  end load_data;
148  /

Package body created.
```

Как видите, тут все немного запутано. Наша подпрограмма создает код, который будет динамически выполняться. В этом динамически формируемом коде используется оператор **FORALL** для множественной вставки строк из массивов. Поскольку в операторе **EXECUTE IMMEDIATE** можно использовать только типы данных SQL, пришлось заранее создать в базе данных соответствующие типы. После этого необходимо динамически выполнять оператор:

```
begin
   forall i in 1 .. :n
       insert into t (a,b,c) values (:a(I), :b(I), :c(I));
end;
```

подставляя в него количество вставляемых строк и три массива значений. Как будет показано ниже, обработка массивов ускоряет вставку во много раз. Необходимо решить, стоит ли это ускорение того, чтобы отказаться от простоты написания подпрограммы с помощью встроенного динамического SQL при отсутствии массивов. Конечно, трудно что-то противопоставить одной строке кода! Если речь идет о программе одноразового использования, для которой производительность несущественна, я бы выбрал самый простой способ. Если речь идет о многократно используемой подпрограмме, которую будут использовать достаточно долго, я бы выбрал пакет **DBMS_SQL,** если скорость работы имеет значение и количество связываемых переменных заранее не известно, и — встроенный динамический SQL, если производительность приемлема, а количество связываемых переменных хорошо известно.

Наконец, не стоит забывать о результатах, представленных в главе 10, — там было показано, что желательно сокращать количество мягких разборов. Пакет DBMS_SQL позволяет это сделать, а встроенный динамический SQL — нет. Необходимо хорошо представлять себе, что именно надо сделать, и выбирать соответствующий подход. Если пишется программа загрузки данных, которую запускают раз в день, и при этом запросы анализируются всего несколько сотен раз, встроенный динамический SQL прекрасно подходит. С другой стороны, если пишется подпрограмма, использующая один и тот же динамический SQL-оператор десятки раз и выполняемая одновременно десятками пользователей, имеет смысл использовать средства пакета **DBMS_SQL,** чтобы можно было проанализировать запрос один раз, а затем только выполнять.

Я выполнил представленные ранее подпрограммы с помощью следующего тестового кода (помните, мы работаем в однопользовательской системе!):

```
create table t (a int, b date, c varchar2(15));

alter session set sql_trace=true;
truncate table t;
```

```
exec load_data.dbmssql_array('t', 50, 10000);

truncate table t;
exec load_data.native_dynamic_array('t', 50, 10000);

truncate table t;
exec load_data.dbmssql_noarray('t', 10000)

truncate table t;
exec load_data.native_dynamic_noarray('t', 10000)
```

В отчете TKPROF можно обнаружить следующее:

```
BEGIN load_data.dbms_sql_array('t', 50, 10000); END;
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.00	0	0	0	0
Execute	1	2.58	2.83	0	0	0	1
Fetch	0	0.00	0.00	0	0	0	0
total	**2**	**2.59**	**2.83**	**0**	**0**	**0**	**1**

```
BEGIN load_data.native_dynamic_array('t', 50, 10000); END;
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	2.39	2.63	0	0	0	1
Fetch	0	0.00	0.00	0	0	0	0
total	**2**	**2.39**	**2.63**	**0**	**0**	**0**	**1**

Общие профили выполнения очень близки: 2,59 и 2,30 секунд процессорного времени. Различие — в деталях. Если вы обратили внимание, в представленном ранее коде я сделал каждый оператор вставки уникальным, добавив псевдонимы таблиц **Q1, Q2, Q3 и Q4.** Благодаря этому можно определить, сколько раз анализировался каждый оператор. В подпрограмме на основе пакета **DBMS_SQL** и массивов использовался псевдоним Q1, а в подпрограмме со встроенным динамическим SQL — псевдоним Q4. Получены следующие результаты:

```
insert into t q1 (a, b, c) values (:a, :b, :c)
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.01	0	0	0	0

и:

```
begin
    forall i in 1 .. :n
       insert into t q4 (a, b, c) values (:a(i), :b(i), :c(i));
end;
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	200	0.10	0.07	0	0	0	0

```
INSERT INTO T Q4 (A,B,C) VALUES (:Ы,:Ь2,:Ь3)
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	200	0.07	0.04	0	0	0	0

Как видите, подпрограмме, использующей средства пакета **DBMS_SQL,** хватило всего одного разбора, а вот при использовании встроенного динамического SQL анализировать операторы пришлось 400 раз. В загруженной системе, где одновременно работает множество пользователей, это может существенно снизить производительность. Поскольку избежать избыточных разборов можно и соответствующий код с использованием средств пакета **DBMS_SQL** не намного сложнее, я считаю оптимальным при решении подобного рода задач использовать **DBMS_SQL.** Хотя код и сложнее, но для обеспечения более высокой масштабируемости я бы использовал именно его.

Если сравнить результаты процедур, не обрабатывающих массивы, оказывается, что они существенно хуже:

```
BEGIN load_data.dbmssql_noarray('t', 10000); END;
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	7.66	7.68	0	0	0	1
Fetch	0	0.00	0.00	0	0	0	0
total	2	7.66	7.68	0	0	0	1

```
BEGIN load_data.native_dynamic_noarray('t', 10000); END;
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	6.15	6.25	0	0	0	1
Fetch	0	0.00	0.00	0	0	0	0
total	2	6.15	6.25	0	0	0	1

Несомненно, имеет смысл использовать встроенный динамический SQL. Но и без массивов я все равно использовал бы средства пакета **DBMS_SQL. И** вот почему:

```
insert into t q2 (a, b, c) values (:a, :b, :c)
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0

```
insert into t q3 (a, b, c) values (:a, :b, :c)
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	10000	1.87	1.84	0	0	0	0

Оказывается, что при использовании встроенного динамического SQL пришлось выполнить 10000 мягких разборов, и лишь один — при использовании пакета **DBMS_SQL В** многопользовательской среде реализация на основе пакета **DBMS_SQL** окажется намного более масштабируемой.

Аналогичные результаты можно получить и при обработке множества строк, выдаваемых по динамически формируемому запросу. Обычно данные можно выбирать из курсорных переменных массивами, но только из *строго типизированных*. Это такие курсорные переменные, структура которых известна на этапе компиляции. Встроенный динамический SQL поддерживает использование только слабо типизированных курсорных переменных и поэтому не поддерживает множественную выборку, **BULK COLLECT**. Если попытаться выполнить оператор **BULK COLLECT** для динамически открытой курсорной переменной, будет получено сообщение об ошибке:

```
ORA-01001: Invalid Cursor
```

Вот сравнение двух процедур, выбирающих и подсчитывающих все строки из представления **ALL_OBJECTS**. Процедура, использующая средства пакета **DBMS_SQL** и обрабатывающая массивы, работает почти вдвое быстрее:

```
scott@TKYTE816> create or replace procedure native_dynamic_select
  2 as
  3     type rc is ref cursor;
  4     l_cursor rc;
  5     l_oname  varchar2(255);
  6     l_cnt    number := 0;
  7     l_start  number default dbms_utility.get_time;
  8 begin
  9     open l_cursor for 'select object_name from all_objects';
 10
 11     loop
 12         fetch l_cursor into l_oname;
 13         exit when l_cursor%notfound;
 14         l_cnt := l_cnt+1;
 15     end loop;
 16
 17     close l_cursor;
 18     dbms_output.put_line(L_cnt || ' rows processed');
 19     dbms_output.put_line
 20     (round((dbms_utility.get_time-l_start)/100, 2) || ' seconds');
 21 exception
 22     when others then
 23         if (l_cursor%isopen)
 24         then
 25             close l_cursor;
 26         end if;
 27         raise;
 28 end;
 29 /

Procedure created.

scott@TKYTE816> create or replace procedure dbms_sql_select
  2 as
  3     l_theCursor     integer default dbms_sql.open_cursor;
  4     l_columnValue   dbms_sql.varchar2_table;
  5     l_status        integer;
  6     l_cnt           number := 0;
```

```
 7      l_start  number default dbms_utility.get_time;
 8  begin
 9
10      dbms_sql.parse(l_theCursor,
11           'select           object_name from all_objects',
12                     dbms_sql.native);
13
14     dbms_sql.define_array(l_theCursor, 1, l_columnValue, 100, 1);
15      l_status := dbms_sql.execute(l_theCursor);
16      loop
17          l_status := dbms_sql.fetch_rows(l_theCursor);
18          dbms_sql.column_value(l_theCursor,1,l_columnValue);
19
20         l_cnt := l_status+l_cnt;
21          exit when l_status <> 100;
22      end loop;
23      dbms_sql.close_cursor(l_theCursor);
24     dbms_output.put_line(L_cnt || ' rows processed');
25      dbms_output.put_line
26     (round((dbms_utility.get_time-l_start)/100, 2 ) || ' seconds');
27  exception
28      when others then
29          dbms_sql.close_cursor(l_theCursor);
30          raise;
31  end;
32  /

Procedure created.

scott@TKYTE816> set serveroutput on

scott@TKYTE816> exec native_dynamic_select
19695 rows processed
1.85 seconds

PL/SQL procedure successfully completed.

scott@TKYTE816> exec native_dynamic_select
19695 rows processed
1.86 seconds

PL/SQL procedure successfully completed.

scott@TKYTE816> exec dbms_sql_select
19695 rows processed
1.03 seconds

PL/SQL procedure successfully completed.

scott@TKYTE816> exec dbms_sql_select
19695 rows processed
1.07 seconds

PL/SQL procedure successfully completed.
```

Снова приходится выбирать между производительностью и простотой кода. Для обработки массивов средствами пакета **DBMS_SQL** необходимо написать намного боль-

ше кода, чем при использовании встроенного динамического SQL, но производительность при этом существенно возрастает.

Проблемы

Как и в случае любого средства, при использовании динамического SQL есть ряд нюансов, которые необходимо учитывать. В этом разделе мы рассмотрим их последовательно. При использовании динамического SQL в хранимых процедурах возникает три основных проблемы:

- нарушается цепочка зависимостей;
- код становится более "хрупким";
- настройка, обеспечивающая предсказуемое время выполнения, существенно усложняется.

Нарушение цепочки зависимостей

Обычно при компиляции процедуры в базе данных все объекты, на которые она ссылается, а также все объекты, ссылающиеся на нее, регистрируются в словаре данных. Например, я создам функцию:

```
ops$tkyte@DEV816> create or replace function count_emp return number
  2 as
  3         l_cnt number;
  4 begin
  5        select count(*) into l_cnt from emp;
  6         return l_cnt;
  7 end;
  8 /

Function created.

ops$tkyte@DEV816> select referenced_name, referenced_type
  2    from user_dependencies
  3   where name = 'COUNT_EMP'
  4     and type = 'FUNCTION'
  5 /

REFERENCED_NAME                                                  REFERENCED_T

STANDARD                                                         PACKAGE
SYS_STUB_FOR_PURITY_ANALYSIS                                     PACKAGE
EMP                                                              TABLE

3 rows selected.
```

Сравним это с тем, что зарегистрировано при последнем создании использующей встроенный динамический SQL функции GET_ROW_CNTS:

```
ops$tkyte@DEV816> select referenced_name, referenced_type
  2    from user_dependencies
  3   where name = 'GET_ROW_CNTS'
```

```
  4     and type = 'FUNCTION'
  5  /

REFERENCED_NAME                                                  REFERENCED_T

STANDARD                                                         PACKAGE
SYS_STUB_FOR_PURITY_ANALYSIS                                     PACKAGE
2 rows selected.
```

Для функции, статически ссылающейся на таблицу **EMP,** эта ссылка зарегистрирована в таблице зависимостей. Для функции же с динамическим SQL — нет, поскольку она не зависит от таблицы **EMP.** В данном случае это вообще не проблема, поскольку использование динамического SQL дает другое существенное преимущество — возможность определить количество строк в таблице. Ранее, однако, мы иногда использовали динамический SQL без особой нужды, и эта нарушенная цепочка зависимостей — очень плохой побочный эффект. Необходимо знать, на какие объекты ссылаются процедуры и где они используются. При использовании динамического SQL такие взаимосвязи не отслеживаются.

"Хрупкость" кода

При использовании только статического SQL можно быть уверенным, что если ух программа успешно скомпилирована, во встроенных в нее SQL-операторах нет синтаксических ошибок, т.к. при компиляции все проверяется. При использовании динамического SQL его корректность будет определена только на этапе выполнения. Более того, поскольку SQL-операторы создаются динамически, необходимо проверять возможные ветвления кода, чтобы убедиться в корректности генерируемого SQL-кода. Если при определенных входных данных генерируется корректный SQL-оператор, это еще не означает, что все будет работать при любых входных данных. Это верно для **любого** кода, но использование динамического SQL делает код несколько уязвимым.

Динамический SQL позволяет сделать многое, что по-другому сделать нельзя, но по возможности надо использовать статический SQL. Он выполняется быстрее, требует меньше ресурсов и менее уязвим.

Сложность настройки

Это неочевидно, но приложение, динамически создающее запросы, настраивать сложно. Обычно можно получить полный список запросов, используемых приложением, выделить те из них, которые влияют на производительность, и бесконечно их настраивать. Если набор выполняемых приложением запросов не известен до того, как приложение начнет работать, нельзя точно узнать, какой будет его производительность. Предположим, создается хранимая процедура, динамически строящая запрос на основе введенных пользователем в Web-форме данных. Если не протестировать все возможные запросы, которые могут быть сгенерированы этой процедурой, нельзя понять, все ли необходимые индексы созданы и можно ли вообще считать настройку системы законченной. Даже при небольшом количестве столбцов (скажем, пяти) могут быть заданы десятки условий. Это не означает, что следует отказаться от использования динамичес-

кого SQL, но будьте готовы к такого рода проблемам, если выполняются запросы, о возможности генерирования которых системой вы даже не задумывались.

Резюме

В этой главе мы детально изучили использование динамического SQL в хранимых процедурах; рассмотрели различия между его реализацией средствами встроенного динамического SQL и пакета **DBMS_SQL;** выяснили, когда использовать тот или иной подход. Оба подхода имеют свои преимущества и назначение.

Динамический SQL позволяет создавать процедуры, реализовать которые иначе просто невозможно, — универсальные утилиты выгрузки и загрузки данных и т.п. На сайте издательства Wrox вы можете найти программы, использующие динамический SQL, утилиту для загрузки файлов dBASE III в базу данных Oracle с помощью PL/SQL, сценарий для печати результатов выполнения запросов в SQL*Plus по столбцам (об этом мы поговорим подробно в главе 23), программы для транспонирования результирующих множеств и многое другое.

17

interMedia

interMedia — это набор средств, тесно интегрированных в СУБД Oracle и обеспечивающих загрузку в базу данных и безопасное управление разнородной информацией (rich content), а также доступ к ней в приложениях. Подобная информация широко используется в большинстве современных Web-приложений и включает текст, данные, изображения, аудио- и видеофайлы.

Эта глава посвящена одному из моих любимых компонентов interMedia: interMedia Text. Я считаю, что технология interMedia Text обычно используется мало. Это происходит из-за недостаточного понимания ее сути и возможностей. Большинству специалистов известны только общие сведения о возможностях interMedia; еще они знают, как обеспечить поддержку работы с текстом для своих таблиц. При ближайшем же рассмотрении оказывается, что interMedia Text — замечательное и уникальное средство СУБД Oracle.

После краткого обзора истории interMedia мы:

- обсудим использование компонента interMedia Text, в частности, для поиска текста, индексирования данных из множества различных источников и поиска в приложениях XML;
- кратко рассмотрим, как реализованы соответствующие возможности в СУБД;
- рассмотрим ряд особенностей компонента interMedia Text: индексирование, использование оператора ABOUT и поиска в разделах.

Краткий исторический экскурс

В ходе разработки большого проекта в 1992 году я впервые столкнулся с компонентом interMedia Text, или, как он тогда назывался, SQL*TextRetrieval. В это время одной из моих задач была интеграция множества различных СУБД для создания большой распределенной сети баз данных. Одна из этих СУБД была настолько "закрытой", насколько это вообще возможно. Она не обеспечивала SQL-интерфейс для управления базой данных и доступа к текстовым данным. Мы должны были создать для нее SQL-интерфейс.

Примерно в середине работы над проектом наш консультант по технологиям Oracle предоставил информацию о следующем поколении программного продукта Oracle SQL*TextRetrieval, которое должно было называться TextServer3. Одно из преимуществ TextServer3 состояло в высокой степени оптимизации для работы в клиент/серверной среде. Кроме того, в составе TextServer3 предлагался несколько заумный интерфейс на языке C, но теперь я, по крайней мере, мог хранить все текстовые данные в базе данных Oracle и иметь при этом возможность обращаться к другим данным в той же базе данных с помощью SQL-операторов. Мне это понравилось.

В 1996 году корпорация Oracle выпустила следующее поколение продукта TextServer под названием ConText Option, которое существенно отличалось от предыдущих версий. Не надо было больше хранить тексты и управлять ими через функциональный интерфейс на языке C или в среде Forms. Все можно было сделать в SQL. Компонент ConText Option предоставил множество PL/SQL-процедур и пакетов, позволяющих сохранять текст, создавать индексы, выполнять запросы, выполнять операции сопровождения для индексов и т.п., и для этого больше не требовалось писать ни одной строки кода на языке C. Среди многих преимуществ ConText Option, по моему мнению, два были наиболее существенными. Первое, и самое главное — ConText Option перестал быть лишь слабо интегрированным, периферийным компонентом СУБД. Он поставлялся в составе СУБД Oracle7 как отдельно лицензируемый необязательный компонент СУБД и был интегрирован в Oracle7. Второе преимущество состояло в том, что компонент ConText Option не только выполнял стандартную процедуру поиска текста, но и поддерживал лингвистический анализ текстов и документов, что позволило разработчикам создавать системы, "читающие между строк" и реально учитывающие общий смысл текстов. Не забывайте также, что все это было доступно через SQL, т.е. существенно упрощало использование этих развитых средств.

Одним из существенных усовершенствований СУБД Oracle8i является стройная система расширения возможностей. На основе поддерживаемых служб разработчики получили средства создания специализированных, сложных типов данных, а также возможность организовывать собственные базовые службы СУБД для поддержки этих типов данных. В рамках этой системы расширения можно создавать новые типы индексов, использовать специализированные методы сбора статистической информации, а также интегрировать в СУБД Oracle специализированные функции оценки стоимости и избирательности методов доступа. На основе этой информации оптимизатор запросов может разумно и эффективно обращаться к новым типам данных. Создававшая ConText Option команда разработчиков оценила значимость этой системы расширения и занялась со-

зданием современного продукта, interMedia Text, который впервые появился в составе Oracle8i в 1999 году.

Использование компонента interMedia Text

Компонент interMedia Text можно использовать в приложениях для многих целей, в том числе:

- **Поиск текста.** Компонент interMedia Text позволяет быстро создать приложение, обеспечивающее эффективный поиск в текстовых данных.
- **Управление разнородными документами.** Компонент interMedia Text позволяет создать приложение, обеспечивающее поиск по документам в различных форматах, в том числе в текстовых файлах, файлах Microsoft Word, Lotus 1-2-3 и Microsoft Excel.
- **Индексирование текста из различных источников данных.** Компонент interMedia Text позволяет создать приложение, управляющее текстовыми данными не только в базе данных Oracle, но и в файловой системе и в сети Internet.
- **Создание приложений, "читающих между строк".** Помимо обеспечения эффективного поиска слов и фраз, компонент interMedia Text позволяет построить "базу знаний" с краткими резюме по каждому документу или проклассифицировать документы по описываемым в них понятиям, а не просто по содержащимся словам.
- **Поиск в приложениях XML.** Компонент interMedia Text предоставляет разработчикам приложений все необходимые средства для создания систем, не только запрашивающих содержимое XML-документов, но и позволяющих выполнять запросы к определенной структуре в XML-документе.

Наличие всех этих функциональных возможностей в СУБД Oracle позволяет при работе с текстовыми данными в полном объеме использовать присущую ей масштабируемость и защиту данных.

Поиск текста

Разумеется, есть много способов поиска текста в базе данных Oracle и без использования компонента interMedia. В следующем примере мы создадим простую таблицу, вставим несколько строк, а затем воспользуемся стандартной функцией **INSTR** и оператором **LIKE** для поиска по текстовому столбцу таблицы:

```
SQL> create table mytext
  2  (id     number primary key,
  3   thetext varchar2(4000)
  4  )
  5  /

Table created.

SQL> insert into mytext
  2  values(1, 'The headquarters of Oracle Corporation is ' ||
```

```
  3             'in Redwood Shores, California.');

1 row created.

SQL> insert into mytext
  2  values(2, 'Oracle has many training centers around the world.');

1 row created.

SQL> commit;

Commit complete.

SQL> select id
  2    from mytext
  3   where instr(thetext, 'Oracle') > 0;
        ID

         1
         2

SQL> select id
  2    from mytext
  3   where thetext like '%Oracle%';
        ID

         1
         2
```

С помощью встроенной функции SQL **INSTR** можно искать вхождения подстроки в строке. С помощью оператора **LIKE** можно также искать строки, соответствующие шаблону. Во многих случаях функция **INSTR** или оператор **LIKE** идеально подходят для решения поставленной задачи, и все прочие средства просто избыточны, особенно при поиске в сравнительно небольших таблицах.

Однако эти методы поиска текста обычно требуют полного просмотра таблицы и огромных ресурсов. Более того, функциональные возможности такого поиска весьма ограничены. Они не помогут, например, если необходимо создать приложение, поддерживающее следующие запросы:

- найти все строки, содержащие слово "Oracle" рядом со словом "Corporation" так, что их разделяет не более двух слов;
- найти все строки, содержащие слово "Oracle" или слово "California", отсортировав результаты по релевантности;
- найти все строки со словами, имеющими корень "train" (например, trained, training, trains);
- выполнить поиск строки в библиотеке документов независимо от регистра символов.

Эти запросы — лишь малая часть того, что нельзя сделать традиционными средствами, но легко делается с помощью компонента interMedia Text. Чтобы продемонстрировать, насколько легко interMedia позволяет отвечать на приведенные выше запросы, необходимо сначала создать индекс interMedia Text по нашему текстовому столбцу:

Чтобы использовать PL/SQL-пакеты компонента interMedia Text, пользователю должна быть предоставлена роль C7XAPP.

```
SQL> create index mytext_idx
  2  on mytext(thetext)
  3  indextype is CTXSYS.CONTEXT
  4  /
Index created.
```

Создав индекс нового типа, CTXSYS.CONTEXT, мы обеспечили возможность эффективного поиска текста для существующей таблицы. Теперь можно использовать множество операторов, поддерживаемых компонентом interMedia Text, для сложной обработки текстовых данных. Следующие примеры демонстрируют использование оператора CONTAINS для ответа на четыре представленных ранее запроса (не обращайте пока внимания на особенности синтаксиса SQL-операторов, поскольку он будет подробно рассмотрен далее):

```
SQL> select id
  2    from mytext
  3  where contains(thetext, 'near((Oracle,Corporation),10)') > 0;
        ID

         1

SQL> select score(1), id
  2    from mytext
  3  where contains(thetext, 'oracle or California', 1) > 0
  4   order by score(1) desc
  5  /
  SCORE(1)         ID

         4          1
         3          2

SQL> select id
  2    from mytext
  3  where contains(thetext, '$train') > 0;

        ID

         2
```

Помимо функциональных возможностей индексы interMedia Text превосходят традиционные реляционные методы поиска текста в столбце и по производительности. Этот абсолютно новый тип индекса в базе данных Oracle предоставляет ценную информацию об индексированных данных, которую оптимизатор может использовать при создании плана выполнения запроса. Поэтому ядро СУБД Oracle получает оптимальный способ доступа к данным столбца, проиндексированным с помощью компонента interMedia Text.

Управление разнородными документами

Помимо возможности индексировать текстовые столбцы в базе данных компонент interMedia Text включает набор фильтров документов для множества форматов. Компонент interMedia Text будет автоматически обрабатывать документы Microsoft Word 2000 для Windows, Microsoft Word 98 для Macintosh, электронные таблицы Lotus 1-2-3, документы в формате Adobe Acrobat PDF и даже файлы презентаций Microsoft PowerPoint. Всего в составе компонента interMedia Text поставляется более 150 фильтров для разных типов файлов и документов.

Наличие тех или иных фильтров зависит от версии Oracle 8i. Например, версии Oracle 8.1.5 и Oracle 8.1.6 были выпущены раньше, чем появился формат Microsoft Word 2000 для Windows. Поэтому в составе компонента interMedia Text версии 8.1.5 или 8.1.6 нет фильтра для этого типа документов, а в составе interMedia Text в Oracle 8.1.7 — уже есть.

Технология фильтрования, включенная в состав interMedia Text, получена по лицензии от корпорации Inso Corporation и с точки зрения точности и эффективности я считаю ее лучшей из имеющихся на рынке. Список текущих поддерживаемых форматов файлов представлен в приложении к руководству *Oracle8i interMedia Text Reference.*

На момент написания этой книги фильтры Inso ***не были*** *доступны на платформе Linux, и перенос их на эту платформу не планировался, что очень печально. Это не означает, что компонент interMedia Text нельзя использовать на платформе Linux, но если используется версия Oracle 8i для Linux, придется либо ограничиться текстовыми и HTML-документами, либо создавать так называемые* пользовательские фильтры, *объекты* ***USER_FILTER.***

Индексирование текста из различных источников данных

Компонент interMedia Text обеспечивает не только хранение файлов в базе данных. *Источник данных* **(datastore object)** interMedia Text позволяет указать, где именно должен храниться текст или данные. Источник данных обеспечивает необходимую для индекса interMedia Text информацию о том, где находятся данные. Эту информацию можно задать только при создании индекса.

Как было показано в предыдущем примере, данные для индекса interMedia Text могут поступать непосредственно из базы данных — храниться в столбце таблицы. Этот источник данных, **DIRECT_DATASTORE,** является стандартным, если явно не указан другой источник. Столбец может быть типа **CHAR, VARCHAR, VARCHAR2, BLOB, CLOB** или **BFILE.** Можно создать индекс interMedia Text и по столбцу типа **LONG** или **LONG RAW,** но с момента выхода версии Oracle 8 эти типы считаются устаревшими, и их не **стоит** использовать во вновь создаваемых приложениях.

Еще один полезный тип источника данных для текста, хранящегося в столбцах таблиц базы данных, — **DETAIL_DATASTORE.** Отношение главный/подчиненный часто

встречается в приложениях. Это отношение задает связь между строкой в главной (родительской) таблице и нулем или более строк в подчиненной таблице и реализуется с помощью требования внешнего ключа в подчиненной таблице, ссылающегося на главную. Счет-фактура — хороший пример отношения главный/подчиненный: обычно одному счету-фактуре соответствует ноль или более строк в подчиненной таблице, описывающей купленные товары. Источник данных типа **DETAIL_DATASTORE** позволяет разработчику учесть эту логическую взаимосвязь таблиц.

Давайте рассмотрим пример. Необходимо создать такую структуру из главной и подчиненной таблицы, чтобы запрос с помощью средств interMedia Text обращался к главной таблице, но фактически данные для interMedia Text брались бы из подчиненной таблицы. Создадим сначала главную и подчиненную таблицы и наполним их данными:

```
SQL> create table purchase_order
  2  (id                  number primary key,
  3   description         varchar2(100),
  4   line_item_body      char(1)
  5  )
  6  /

Table created.

SQL> create table line_item
  2  (po_id            number,
  3   po_sequence      number,
  4   line_item_detail varchar2(1000)
  5  )
  6  /

Table created.

SQL> insert into purchase_order (id, description)
  2  values(1, 'Many Office Items')
  3  /

1 row created.

SQL> insert into line_item(po_id, po_sequence, line_item_detail)
  2  values(1, 1, 'Paperclips to be used for many reports')
  3  /

1 row created.

SQL> insert into line_item(po_id, po_sequence, line_item_detail)
  2  values(1, 2, 'Some more Oracle letterhead')
  3  /

1 row created.

SQL> insert into line_item(po_id, po_sequence, line_item_detail)
  2  values(1, 3, 'Optical mouse')
  3  /

1 row created.

SQL> commit;
```

Обратите внимание: столбец LINE_ITEM_BODY по сути "фиктивный" — он существует, чтобы можно было создать индекс interMedia Text по главной таблице. Я никогда не буду вставлять в него данные. Прежде чем создавать индекс, необходимо задать параметры interMedia Text так, чтобы при создании индекса были найдены индексируемые данные:

```
SQL> begin
  2   ctx_ddl.create_preference('po_pref',   'DETAIL_DATASTORE');
  3   ctx_ddl.set_attribute('po_pref',   'detail_table', 'line_tem' );
  4   ctx_ddl.set_attribute('po_pref',   'detail_key', 'po_id');
  5   ctx_ddl.set_attribute('po_pref',   'detail_lineno', 'po_sequence');
  6   ctx_ddl.set_attribute('po_pref',   'detail_text', 'line_item_detail');
  7  end;
  8  /

PL/SQL procedure successfully completed.
```

Сначала мы создаем пользовательский параметр **PO_PREF**. Это источник данных типа **DETAIL_DATASTORE**, в котором будет храниться вся необходимая информация для поиска данных в подчиненной таблице. В следующих строках мы задаем имя подчиненной таблицы, ключ, по которому выполняется соединение с главной таблицей, порядок следования строк и, наконец, индексируемый столбец.

Теперь создадим индекс и проверим его в работе:

```
SQL> create index po_index on purchase_order(line_item_body)
  2  indextype is ctxsys.context
  3 parameters('datastore po_pref')
  4  /

Index created.

SQL> select id
  2    from purchase_order
  3  where contains(line_item_body, 'Oracle') > 0
  4  /

        ID

         1
```

Хотя индекс создается по столбцу **LINE_ITEM_BODY,** при создании можно задать и столбец главной таблицы **DESCRIPTION.** Помните, однако, что любые изменения этого столбца (этот столбец — не фиктивный) вызовут переиндексацию строки главной таблицы и связанных с ней строк подчиненной таблицы.

Компонент interMedia Text поддерживает также внешние источники данных, в частности файлы, не входящие в базу данных, а также универсальные локаторы ресурсов (Uniform Resource Locators — URLs). Во многих производственных средах файлы обычно хранятся в доступной по сети общей файловой системе. Необязательно хранить тексты и документы приложения в базе данных Oracle. Можно создать источник данных типа **FILE_DATASTORE** — это позволит серверу Oracle управлять только текстовым индексом и не заниматься хранением и защитой файлов. При использовании источника данных **FILE_DATASTORE** не надо хранить текст документа в столбце. Необходимо

хранить ссылку на файл в файловой системе, по которой можно обратиться к файлу **на сервере.** Так что, даже если используется, например, Windows-клиент, а сервер Oracle работает на вашей любимой разновидности ОС UNIX, ссылка на файл должна задаваться по правилам файловой системы ОС UNIX, например **/export/home/tkyte/MyWordDoc.doc.** Учтите, что этот способ доступа к внешним файлам никак не связан с альтернативным способом доступа из базы данных Oracle с помощью данных типа **BFILE.**

Еще один тип источника данных, внешнего по отношению к базе данных, — URL_DATASTORE. Он очень похож на источник данных **FILE_DATASTORE,** но вместо ссылки на файл в файловой системе в столбце таблицы хранится URL. В момент индексации строки компонент interMedia Text фактически прочитает данные по протоколу HTTP. Но и в этом случае сервер Oracle не хранит и не управляет этими данными. Индекс создается на основе профильтрованного содержимого потока данных **HTTP**, а сами выбранные по URL данные не сохраняются. Протокол FTP тоже поддерживается при использовании источника данных **URL_DATASTORE,** так что interMedia Text позволяет индексировать также файлы, доступные для сервера баз данных по FTP. При использовании версии Oracle 8.I.7 или более новой можно также встраивать имя пользователя и пароль для **FTP** непосредственно в строку URL (например, **ftp://** uid:pwd@ftp.bogus.com/tmp/test.doc).

Некоторые думают, что источник данных **URL_DATASTORE** пригоден только для создания поискового робота (кстати, этой возможностью поиска в Web он в исходном виде не обладает). Это неверно. Мои коллеги создали очень большую распределенную систему баз данных с возможностью доступа из сети Internet. Она должна была обеспечить единый универсальный интерфейс поиска текстовых данных во всех задействованных базах. Для этого можно было создать систему индексов interMedia Text по таблицам в каждой базе данных, а затем выполнять декомпозицию запроса на множество распределенных запросов к этим базам данных. Однако они не пошли по пути, ведущему к неоптимальной производительности, а решили выделить один сервер для поддержки компонента interMedia Text и создали все индексы с помощью источника данных **URL_DATASTORE**. В этой системе удаленные базы данных отвечали за вставку адресов URL для новых или изменившихся документов в базу данных, обеспечивающую поиск. Таким образом, при каждом создании нового или изменении существующего документа серверу, обеспечивающему индексацию документов, передается URL для получения содержания этого документа. Индексирующей машине не приходится искать новые и измененные документы: предоставляющие их серверы просто уведомляют ее о новых поступлениях. При этом не только не нужен распределенный запрос, но и создается централизованный источник информации, что упрощает администрирование.

Компонент interMedia Text - часть базы данных Oracle

Одной из наиболее существенных причин для использования компонента interMedia Text вместо решений на базе файловой системы является то, что этот компонент входит в базу данных Oracle. Во-первых, в базе данных Oracle поддерживаются транзакции, а в файловой системе — нет. Целостность данных не нарушается, а свойства **ACID** реляционной базы данных распространяются и на компонент interMedia Text.

*Свойства **ACID** (неделимость, согласованность, изолированность и продолжительность) представлены в главе 4 (в первой части книги — **прим. научн. ред.**)*

Во-вторых, в Oracle для работы с базой данных предлагается язык SQL, и компонент interMedia Text полностью доступен в SQL-операторах. Это позволяет при разработке приложений использовать множество инструментальных средств, "понимающих" язык SQL. При желании (хотя я этого и не рекомендую) можно создать приложение, использующее возможности компонента interMedia Text в электронных таблицах Microsoft Excel, подключаясь к базе данных Oracle через интерфейс ODBC.

Поскольку в течение своей карьеры я часто выполнял функции администратора базы данных, меня очень радует тот факт, что все данные будут содержаться в базе данных Oracle, и при ее резервном копировании будет копироваться также приложение и все его данные. При необходимости можно будет восстановить приложение и его данные по состоянию на любой момент времени. Если используется решение на базе файловой системы, придется проверять, создана ли резервная копия базы данных и соответствующей файловой системы, и надеяться, что в момент копирования они были согласованы.

При использовании компонента interMedia Text для индексирования информации, содержащейся вне базы данных Oracle, однако, необходимо немного изменить стратегию резервного копирования. Если используются источники данных **URL_DATASTORE** или **FILE_DATASTORE,** компонент interMedia Text поддерживает только ссылки на документы, но не сами документы. Документы эти со временем устаревают, удаляются или оказываются недоступными по другим причинам, и это может отрицательно сказаться на работе приложения. Кроме того, при резервном копировании базы данных Oracle уже не происходит полное резервное копирование приложения. Необходимо придумать отдельную стратегию резервного копирования для документов, хранящихся вне базы данных Oracle.

Смысловой анализ

Обратитесь к своей любимой поисковой системе, введите часто встречающееся в сети Internet слово, например 'database', и ждите в ответ огромного количества результатов поиска. Индексирование текстов —очень мощное средство, которое можно использовать во многих приложениях. Но его бывает недостаточно. Это особенно верно для очень больших объемов данных, анализировать которые пользователю сложно. Компонент interMedia Text включает интегрированные средства, позволяющие преобразовать все эти данные в полезную информацию.

В компонент interMedia Text интегрирована расширяемая база знаний, которая используется в ходе индексирования и анализа текста и обеспечивает возможность лингвистического анализа. Можно не только искать текст, но и **анализировать** его смысл. Так что при создании индекса interMedia Text можно дополнительно сгенерировать список *тем* документов. Это позволяет создавать приложения, например, для анализа документов и классификации их по темам, а не по содержащимся словам или фразам.

Когда возможность тематической классификации впервые появилась в базе данных Oracle, я придумал простой тест, чтобы оценить в ее возможности. Я загрузил в таблицу базы данных Oracle около тысячи коротких новостей из различных компьютерных жур-

налов. Затем создал индекс interMedia Text по столбцу, использовавшемуся для хранения текста статей, и сгенерировал список тем для каждой статьи. Выполнив поиск документов, посвященных теме "database", я обнаружил, что в их числе оказались статьи, не содержащие слова "database" и тем не менее отнесенные компонентом interMedia Text к теме "база данных" (database). Сначала я подумал, что это — ошибка в компоненте interMedia Text, но, разобравшись, понял, что обладаю потрясающей возможностью — находить в базе данных текст **по смыслу.** Речь не идет о статистическом анализе или хитроумном способе подсчета вхождений слов — это именно лингвистический анализ текста.

Продемонстрирую эти возможности на примере:

```
SQL> create table mydocs
  2  (id      number primary key,
  3   thetext varchar2(4000)
  4  )
  5  /

Table created.

SQL> create table mythemes
  2  (query_id number,
  3   theme    varchar2(2000),
  4   weight   number
  5  )
  6  /

Table created.

SQL> insert into mydocs(id, thetext)
  2  values(1,
  3   'Go to your favorite Web search engine, type in a frequently
  4  occurring word on the Internet like ''database'', and wait
  5  for the plethora of search results to return.'
  6  )
  7  /

1 row created.

SQL> commit;

Commit complete.

SQL> create index my_idx on mydocs(thetext) indextype is ctxsys.context;

Index created.

SQL> begin
  2      ctx_doc.themes(index_name => 'my_idx',
  3                     textkey    => '1',
  4                     restab     => 'mythemes'
  5      );
  6  end;
  7  /

PL/SQL procedure successfully completed.

SQL> select theme, weight from mythemes order by weight desc;
```

```
THEME                                    WEIGHT

occurrences                                  12
search engines                               12
Internet                                     11
result                                       11
returns                                      11
databases                                    11
searches                                     10
favoritism                                    6
type                                          5
plethora                                      4
frequency                                     3
words                                         3

12 rows selected.
```

PL/SQL-блок берет таблицу, на которую ссылается индекс **MY_IDX,** находит строку со значением **key** = 1 и выбирает проиндексированные данные. Затем эти данные обрабатываются тематическим анализатором. Анализатор генерирует список затронутых в документе тем, присваивая им "вес" (например, статья про банковскую деятельность может касаться тем "деньги", "кредит" и т.п.). Затем информация об этих темах помещается в таблицу **MYTHEMES.**

Если проделать это для всех данных в приложении, пользователи смогут искать не только строки, содержащие определенное слово, но и строки, наиболее близкие по смыслу к определенному тексту.

Учтите, что если предоставить компоненту interMedia Text больше данных для анализа, сгенерированный список тем может оказаться намного точнее, чем для рассмотренного простого предложения.

Учтите также, что я создал столбец **ID** как первичный ключ. Для компонента interMedia Text в Oracle 8i 8.1.6 и более ранних версиях необходимо наличие первичного ключа для таблицы, прежде чем по ней можно будет создавать индекс interMedia Text. В Oracle 8i 8.1.7 и последующих версиях компонент interMedia Text больше не требует наличия первичного ключа при создании индекса.

Поиск в приложениях XML

У меня часто спрашивают, как обеспечить эффективный поиск в документе со встроенной разметкой, например, на языке HTML или XML. К счастью, компонент interMedia Text позволяет очень просо решить эту задачу — за счет использования объектов, называемых *разделами*.

Это решение легко использовать, сочетая возможности анализа XML (XML parsing) и задания разделов в источнике данных **URL_DATASTORE.** Если XML соответствует декларируемым целям, т.е. является **средством взаимодействия** разнородных систем, то разработчик приложения с помощью компонента interMedia Text может легко создать оперативную базу знаний с возможностями поиска данных из различных систем. Полный пример индексирования XML-документов представлен далее в этой главе.

Как работает компонент interMedia Text

В этом разделе описано, как реализован компонент interMedia Text и что дает его использование.

Как уже упоминалось, компонент interMedia Text создан с использованием стандартного механизма расширения Oracle. С помощью соответствующих средств команда разработчиков компонента interMedia Text смогла добавить в СУБД Oracle специфический тип индекса для текста. Внимательней присмотревшись к используемым объектам базы данных, можно "приподнять занавес" и получить представление о том, как реализован этот компонент. Объекты базы данных, составляющие компонент interMedia Text, всегда принадлежат пользователю **CTXSYS**:

```
SQL> connect ctxsys/ctxsys
Connected.

SQL> select indextype_name, implementation_name
  2    from user_indextypes;

INDEXTYPE_NAME                 IMPLEMENTATION_NAME

CONTEXT                        TEXTINDEXMETHODS
CTXCAT                         CATINDEXMETHODS
```

Как видите, в схеме, которой принадлежит компонент interMedia Text, имеется два типа индексов. Один из индексов, **CONTEXT**, знаком большинству пользователей компонента interMedia Text. Второй индекс, **CTXCAT**, — это индекс для каталога, обеспечивающий подмножество возможностей, доступных при использовании индекса **CONTEXT**. Индекс для каталога, появившийся в версии Oracle 8.1.7, идеально подходит для текстовых данных, представляющих собой небольшие фрагменты текста.

```
SQL> select library_name, file_spec, dynamic from user_libraries;

LIBRARY_NAME                   FILE_SPEC                      D

DR$LIB                                                        N
DR$LIBX                        O:\Oracle\Ora81\Bin\oractxx8.dll Y
```

Как видите, с компонентом interMedia Text связаны две библиотеки. DRSLIB не является динамически компонуемой и представляет собой библиотеку проверенного кода в самой СУБД Oracle. DRSLIBX — это разделяемая, динамически компонуемая библиотека, зависящая от соответствующей операционной системы. Поскольку этот запрос был выполнен к базе данных, работающей в среде Windows, имя файла, содержащего эту разделяемую библиотеку, отражает особенности Windows. Если выполнить такой же запрос в среде UNIX, результат будет другим. Эти библиотеки специально предназначены для компонента interMedia Text. Они содержат набор методов, позволяющих ядру СУБД Oracle обрабатывать соответствующие индексы interMedia Text.

```
SQL> select operator_name, number_of_binds from user_operators;

OPERATOR_NAME                  NUMBER_OF_BINDS

CATSEARCH                                   2
```

CONTAINS	**8**
SCORE	**5**

В рамках механизма расширения можно также создавать уникальные объекты базы данных — *операторы*. Оператор используется индексом; с каждым оператором ассоциируется ряд связываний (bindings). Во многом аналогично языку PL/SQL, где можно определять функции с одинаковыми именами, но с различными типами параметров (сигнатурой), механизм расширения позволяет определить оператор, соответствующий различным пользовательским методам, в зависимости от сигнатуры использования.

```
SQL> select distinct method_name, type_name from user_method_params order by
  2  type_name;
```

METHOD_NAME	TYPE_NAME
ODCIGETINTERFACES	CATINDEXMETHODS
ODCIINDEXALTER	CATINDEXMETHODS
ODCIINDEXCREATE	CATINDEXMETHODS
ODCIINDEXDELETE	CATINDEXMETHODS
ODCIINDEXDROP	CATINDEXMETHODS
ODCIINDEXGETMETADATA	CATINDEXMETHODS
ODCIINDEXINSERT	CATINDEXMETHODS
ODCIINDEXTRUNCATE	CATINDEXMETHODS
ODCIINDEXUPDATE	CATINDEXMETHODS
ODCIINDEXUTILCLEANUP	CATINDEXMETHODS
ODCIINDEXOTILGETTABLENAMES	CATINDEXMETHODS
RANK	CTX_FEEDBACK_ITEM_TYPE
ODCIGETINTERFACES	TEXTINDEXMETHODS
ODCIINDEXALTER	TEXTINDEXMETHODS
ODCIINDEXCREATE	TEXTINDEXMETHODS
ODCIINDEXDROP	TEXTINDEXMETHODS
ODCIINDEXGETMETADATA	TEXTINDEXMETHODS
ODCIINDEXTRUNCATE	TEXTINDEXMETHODS
ODCIINDEXUTILCLEANUP	TEXTINDEXMETHODS
ODCIINDEXUTILGETTABLENAMES	TEXTINDEXMETHODS
ODCIGETINTERFACES	TEXTOPTSTATS

```
21 rows selected.
```

После просмотра этих результатов становится понятным, как разработчик может использовать механизм расширения в базе данных Oracle. С каждым типом ассоциированы наборы поименованных методов, которые механизм расширения различает по уникальным именам. Например, методы, связанные с поддержкой индекса - **ODCIIndexInsert, ODCIIndexUpdate** и **ODCIIndexDelete,** вызываются СУБД Oracle при создании, изменении или удалении данных, связанных с индексом. Таким образом, при необходимости вставки новой строки в индекс interMedia Text ядро Oracle вызывает метод, ассоциированный с **ODCIIndexInsert.** Это специально созданная подпрограмма, выполняющая необходимые операции с индексом interMedia Text, а затем уведомляющая СУБД Oracle о завершении обработки.

Ознакомившись с основами реализации компонента interMedia Text, давайте рассмотрим некоторые из объектов базы данных, связываемые с этим специализированным индексом interMedia Text при его создании в базе данных.

```
SQL> select table_name
  2    from user_tables
  3   where table_name like '%MYTEXT%';

TOBLE_NAME

MYTEXT

SQL> create index mytext_idx
  2  on mytext(thetext)
  3  indextype is ctxsys.context
  4  /

Index created.

SQL> select table_name
  2    from user_tables
  3   where table_name like '%MYTEXT%';

TABLE_NAME

DR$MYTEXT_IDX$I
DR$MYTEXT_IDX$K
DR$MYTEXT_IDX$N
DR$MYTEXT_IDX$R
MYTEXT
```

Мы начали сеанс SQL*Plus с запроса к представлению **USER_TABLES,** выбирающего все имена таблиц, содержащие подстроку **MYTEXT.** После создания таблицы **MYTEXT** и индекса interMedia Text по столбцу этой таблицы оказывается, что имя пяти таблиц, включая исходную, содержит эту подстроку.

Таким образом, при создании индекса interMedia Text автоматически создается еще четыре таблицы. Имена этих таблиц всегда будут иметь префикс **DRS,** за которым следует имя созданного индекса и один из суффиксов — **$I**, **$K**, **$N** или **$R**. Соответствующие таблицы всегда создаются в той же схеме, что и индекс interMedia Text. Давайте подробнее рассмотрим их структуру.

```
SQL> desc dr$mytext_idx$i;
 Name                                      Null?    Type

 TOKEN_TEXT                                NOT NULL VARCHAR2(64)
 TOKEN_TYPE                                NOT NULL NUMBER(3)
 TOKEN_FIRST                               NOT NULL NUMBER(10)
 TOKEN_LAST                                NOT NULL NUMBER(10)
 TOKEN_COUNT                               NOT NULL NUMBER(10)
 TOKEN_INFO                                         BLOB

SQL> desc dr$mytext_idx$k;
 Name                                      Null?    Type

 DOCID                                              NUMBER(38)
 TEXTKEY                                   NOT NULL ROWID
```

```
SQL> desc dr$mytext_idx$n;
 Name                                      Null?    Type

 NLT_DOCID                                 NOT NULL NUMBER(38)
 NLT_MARK                                  NOT NULL CHAR(1)

SQL> desc dr$mytext_idx$r;
 Name                                      Null?    Type

 ROW_NO                                             NUMBER(3)
 DATA                                               BLOB
```

Для каждого индекса interMedia Text создается набор таблиц с подобной структурой. Таблица лексем, **DR$MYTEXT_IDX$I**, — это основная таблица индекса interMedia Text. Эта таблица используется для хранения каждой проиндексированной лексемы и битовой карты с установленными битами для всех документов, содержащих эту лексему. В этой таблице хранится и другая двоичная информация для оценки близости лексем в тексте. Обратите внимание, что я умышленно использую термин "лексема" в этом абзаце, поскольку компонент interMedia Text позволяет индексировать тексты на языках с иероглифической письменностью, включая китайский, японский и корейский. Было бы некорректно говорить об использовании таблицы **DR$I** для индексирования "слов".

Таблицы **DR$K** и **DR$R** по сути поддерживают соответствие между идентификаторами строк (**ROWID**) и идентификаторами документов.

Последняя таблица, **DRSN**, или "таблица отсутствующих строк", используется для поддержки списка удаленных документов/строк. При удалении строки из таблицы, по которой создан индекс interMedia Text, физическое удаление информации об этой строке из индекса interMedia Text откладывается. В этой служебной таблице записываются идентификаторы документов из удаленных строк для последующего удаления при следующей перестройке или оптимизации индекса.

Учтите также, что таблицы **DR$K** и **DR$N** создаются как организованные по индексу. Обращения к этим таблицам в коде компонента interMedia Text обычно затрагивают оба столбца таблиц. Для повышения эффективности и сокращения объема ввода-вывода эти таблицы организуются по индексу.

Подводя итоги этого раздела, хочу подчеркнуть, что, хотя и интересно разобраться, как компонент interMedia Text реализован с помощью механизма расширения Oracle, это вовсе не обязательно для эффективного использования interMedia Text. Многие разработчики создавали весьма сложные приложения с использованием компонента interMedia Text, ничего не зная о назначении создаваемых таблиц.

Индексирование с помощью interMedia Text

Используя простую таблицу, созданную в предыдущем разделе, давайте по шагам пройдем процесс вставки текста, чтобы увидеть момент фактического выполнения соответствующих изменений компонентом interMedia Text:

```
SQL> delete from mytext;

2 rows deleted.
```

```
SQL> insert into mytext(id, thetext)
  2  values(1, 'interMedia Text is quite simple to use');

1 row created.

SQL> insert into mytext(id, thetext)
  2  values(2, 'interMedia Text is powerful, yet easy to learn');

1 row created.

SQL> commit;

Commit complete.
```

Итак, можно ли сейчас по запросу поиска **Text** получить обе строки таблицы? Может быть. Если индекс interMedia Text не синхронизирован, то выполненные изменения в нем еще не учтены. Синхронизация индекса означает выполнение всех ожидающих учета изменений. Как же определить, есть ли в индексе interMedia Text изменения, ожидающие учета?

```
SQL> select pnd_index_name, pnd_rowid from ctx_user_pending;

PND_INDEX_NAME                 PND_ROWID

MYTEXT_IDX                     AAAGF1AABAAAIV0AAA
MYTEXT_IDX                     AAAGF1AABAAAIV0AAB
```

Выполняя запрос к представлению **CTX_USER_PENDING,** можно определить, что ожидают изменения две строки индекса interMedia Text с именем **MYTEXT_IDX.** Представление **CTX_USER_PENDING** создано по принадлежащей пользователю CTXSYS таблице **DRSPENDING.** При любой вставке строки в таблицу **MYTEXT** в таблицу **DRSPENDING** будет вставляться строка для индекса **MYTEXT_IDX** interMedia Text. Обе вставки выполняются в одной физической транзакции, поэтому, если транзакция, вставившая строку в таблицу **MYTEXT,** будет отменена, произойдет также отмена вставки в таблицу **DRSPENDING.**

Есть три различных способа синхронизации индекса interMedia Text. Эта синхронизация может выполняться в различных условиях и по разным причинам. Позже я скажу о том, когда предпочтительнее использовать тот или иной метод.

Простейший метод синхронизации индекса — запуск программы **ctxsrv.** Эта программа работает аналогично демонам в ОС UNIX. Программа запускается, работает в фоновом режиме и время от времени автоматически синхронизирует индекс. Этот метод рекомендуется использовать при работе с небольшим количеством (до 10000) строк, каждая из которых содержит небольшой объем текста.

Другой метод синхронизации индекса — выполнение оператора **ALTER INDEX.** Можно организовать очередь изменений, ожидающих выполнения, а затем построить и выполнить пакет действий по синхронизации индекса. Во многих случаях это лучший метод синхронизации индекса, обеспечивающий минимальную фрагментацию. Для синхронизации индекса используется следующий оператор:

```
alter index [схема.]индекс rebuild [online]
            parameters('sync [memory объем_памяти]')
```

Имеет смысл пересоздавать индекс в оперативном режиме (online), чтобы он оставался доступным в процессе синхронизации. Кроме того, можно задать объем исполь-

зуемой при этом памяти. Чем больше памяти выделено процессу синхронизации, тем большим может быть пакет индексируемых изменений и тем меньше окажется в итоге индекс interMedia Text.

Хотя многие и сочтут третий метод синхронизации индекса одноразовым, я настаиваю, что простое пересоздание индекса тоже является методом синхронизации. При выполнении оператора CREATE INDEX для создания индекса типа CONTEXT будет создан индекс и проиндексированы все данные столбцов, по которым он создается. При этом часто действия выполняются циклически: в таблице есть данные, мы создаем индекс, а затем добавляем новые строки. Поскольку изменения, связанные с добавлением новых строк, не выполняются до момента синхронизации индекса, многие приходят к выводу, что единственный способ поддержать актуальность индекса — удалить и создать его заново! Индекс действительно синхронизируется, но такой метод крайне неэффективен, и я не рекомендую его использовать.

Семантика языка SQL принципиально не позволяет двум пользователям одновременно выполнять оператор ALTER INDEX REBUILD для одного и того же индекса, но ничто не мешает пересоздавать или синхронизировать одновременно несколько индексов interMedia Text.

Продолжая пример, синхронизируем индекс:

```
SQL> alter index mytext_idx rebuild online parameters('sync memory 20M');

Index altered.

SQL> select pnd_index_name, pnd_rowid from ctx_user_pending;

no rows selected
```

Теперь индекс синхронизирован, и можно выполнять запрос, использующий его:

```
SQL> select id
  2    from mytext
  3   where contains(thetext, 'easy') > 0
  4   /

        ID

         2
```

Просмотрим данные в одной из служебных таблиц, созданных автоматически при создании индекса interMedia Text:

```
SQL> select token_text, token_type from dr$mytext_idx$i;

TOKEN_TEXT                                                    TOKEN_TYPE

EASY                                                                   0
INTERMEDIA                                                             0
LEARN                                                                  0
POWERFUL                                                               0
QUITE                                                                  0
SIMPLE                                                                 0
TEXT                                                                   0
USE                                                                    0
YET                                                                    0
```

```
interMedia Text                                                    1
learning                                                           1

TOKEN_TEXT                                                TOKEN_TYPE


powerfulness                                                       1
simplicity                                                         1
```

Выполняя запрос к таблице **DR$I,** соответствующей индексу **MYTEXT_IDX,** мы можем оценить, какая часть информации обработана компонентом interMedia Text при синхронизации индекса.

Во-первых, обратите внимание, что многие значения в столбце **TOKEN_TEXT** целиком состоят из прописных букв. Это реальные слова из текста, переведенные в верхний регистр. При необходимости можно потребовать от компонента interMedia Text создавать индекс по словам с учетом регистра при выполнении оператора **CREATE INDEX.**

Обратите также внимание, что некоторые лексемы, для которых в столбце **TOKEN_TYPE** находится значение 1, хранятся в смешанном регистре. Что еще важнее, ни в одной из строк таблицы **MYTEXT** нет слов **"simplicity"** и **"learning".** Так откуда же взялись эти данные? При разбиении *лексическим анализатором* блока текста на английском языке на лексемы стандартным действием является добавление в индекс информации о тематике текста. Таким образом, каждая строка со значением **TOKEN_TYPE=1** — это тема, сгенерированная компонентом interMedia Text в процессе лингвистического анализа.

Наконец, нельзя не заметить **отсутствия** некоторых слов в этой таблице. Слова **is** и to не значатся среди лексем в таблице индекса, хотя и входили в исходные данные проиндексированной таблицы. Они являются *стоп-словами* **(stopwords)** и не включаются в индекс как лишняя информация. Эти слова часто многократно встречаются в большинстве текстов на английском и по сути являются "шумом". Слова is, to и около 120 других входят в стандартный *список стоп-слов* **(stoplist)** для английского языка, который и используется по умолчанию при создании индекса. Корпорация Oracle включила в состав компонента interMedia Text стандартные списки стоп-слов более чем для 40 языков. Помните, что список стоп-слов использовать не обязательно, можно создавать и использовать собственные специализированные списки.

Хочу завершить этот раздел предупреждением. Хотя весьма интересно разобраться, как именно устроен компонент interMedia Text, особенно посмотреть, какие лексемы генерируются при создании индекса, **не пытайтесь** создавать другие объекты базы данных, использующие внутренние структуры индекса. В частности, не создавайте представления для таблицы **DR$MYTEXT_IDX$I** или триггеры по таблице **DR$MYTEXT_IDX$K.** Структура реализации может измениться и скорее всего изменится уже в следующих версиях компонента.

Оператор ABOUT

С появлением оператора **ABOUT** в Oracle стало намного проще выполнять тематический анализ в запросах, да и точность результатов, выдаваемых по таким запросам, существенно увеличилась. Для текстов на английском языке оператор **ABOUT** обеспе-

чивает поиск всех строк, соответствующих нормализованному представлению искомого понятия. Как я уже говорил, по умолчанию для текстов на английском языке информация о тематике включается в индекс. Эта информация о тематике текста в индексе будет использоваться для поиска других строк, в которых затрагиваются близкие понятия. Если почему-либо было решено не генерировать информацию о тематике текста при создании индекса, оператор ABOUT будет выполнять простой поиск соответствующих лексем.

```
SQL> select id from mytext where contains(thetext, 'about(databases)') > 0;

no rows selected
```

Как и ожидалось, в таблице нет строк, посвященных понятию **"databases".**

```
SQL> select id from mytext where contains(thetext,'about(simply)') > 0;

        ID

         1
```

Есть одна строка, связанная с понятием "просто" (simply). Если точнее, есть одна строка, содержащая понятия-синонимы нормализованной версии слова simply. Чтобы показать это, выполним:

```
SQL> select id from mytext where contains(thetext,'simply') > 0;

no rows selected
```

При удалении оператора **ABOUT** из запроса ни одна строка не возвращается — в столбец **thetext** не входит слово **simply.** Имеется, однако, одна строка, в которой используемые понятия соответствуют нормализованному корню слова **simply.**

Связанное со словом понятие — не то же самое, что лингвистический корень слова. *Оператор получения основы* (stem operator — $) компонента interMedia Text позволяет искать инфлекционные (inflectional) или производные формы слова. Таким образом, поиск по основе слова **health** может дать в результате документы, содержащие слово healthy. Поиск синонимов понятия **health** (здоровье) с помощью оператора **ABOUT** может также вернуть документы, содержащие слово **wellness.**

Оператор **ABOUT** очень легко включить в приложения для использования всей мощи средств генерации тематической информации и лингвистического анализа. Оператор **ABOUT** позволяет обеспечить в приложении не только поиск введенных пользователем слов, но и поиск связанных с ними понятий. Это действительно мощное средство.

Поиск в разделах

Последняя тема, которую я хочу подробно рассмотреть, — *поиск в разделах.* Разделы обеспечивают избирательный доступ запроса к документу и могут существенно повысить точность запросов. Раздел может представлять собой не что иное, как заданную разработчиком последовательность символов, отмечающую начало и конец логической единицы в документе. Популярность стандартных языков разметки, таких как HTML и XML, позволяет продемонстрировать всю мощь средств поиска в разделах, предлагаемых компонентом interMedia Text.

Типичный документ содержит общеупотребительные логические элементы, образующие его структуру. У большинства документов есть название, может быть заголовок, может быть шаблонная информация, основной текст, содержание, предметный указатель, приложения и т.д. Все это — логические единицы, образующие структуру документа.

В качестве примера ситуации, когда необходим поиск в разделах документов, рассмотрим гипотетическое хранилище документов для Министерства обороны. Может понадобиться найти в хранилище документы, содержащие фразу "Ракета Hellfire". Но может быть еще важнее найти документы, содержащие фразу "Ракета Hellfire" в заголовке документа или, например, в предметном указателе. Компонент interMedia Text позволяет разработчику приложения задать последовательность символов, выделяющую эти логические разделы структуры документа. Кроме того, компонент interMedia Text поддерживает поиск текста в заданных таким образом логических разделах.

В компоненте interMedia Text для задания логических единиц или группировки текста в документе используется понятие "разделы". Набор идентификаторов разделов образует группу разделов, и именно группу разделов можно указать при создании индекса interMedia Text.

Язык разметки гипертекста (Hypertext Markup Language — HTML) тоже первоначально создавался как способ организации структуры документа, но быстро стал языком, частично описывающим структуру, а частично — внешний вид документа. Тем не менее в состав interMedia Text стандартно входят компоненты, позволяющие создать раздел логической структуры документа из каждого тега разметки, содержащегося в документе.

Аналогично, поддержка языка XML тоже встроена в компонент interMedia Text, начиная с версии Oracle 8.1.6. Для XML-документа можно легко (при желании — автоматически) задать разделы, соответствующие каждому заданному в нем XML-элементу.

Давайте сначала рассмотрим следующий пример документа на языке **HTML**:

```
SQL> create table my_html_docs
  2  (id number primary key,
  3  html_text varchar2(4000))
  4  /

Table created.

SQL> insert into my_html_docs(id, html_text)
  2  values(1,
  3  '<html>
  4 <titleX>racle Technology</title>
  5  <body>This is about the wonderful marvels of 8i and 9i</body>
  6  </html>')
  7  /

1 row created.

SQL> commit;

Commit complete.
SQL> create index my_html_idx on my_html_docs(html_text)
  2  indextype is ctxsys.context
```

```
  3  /

Index created.
```

Теперь можно искать строку по проиндексированному в HTML-документе слову.

```
SQL> select id from my_html_docs
  2   where contains(html_text, 'Oracle') > 0
  3  /

        ID

         1

SQL> select id from my_html_docs
  2   where contains(html_text, 'html') > 0
  3  /

        ID

         1
```

Легко создать запрос для поиска всех строк, содержащих слово Oracle, но в полученном решении очевидны два недостатка. Во-первых, элементы разметки индексировать не надо, поскольку они встречаются постоянно и во всех документах и не являются частью содержимого документа. Во-вторых, мы, конечно, можем искать слова в HTML-документе, но без учета структурных элементов, в которых они содержатся. Мы знаем, что где-то в тексте документа есть строка, содержащая слово Oracle, но это может быть заголовок, основной текст, колонтитул и т.п.

Предположим, в приложении необходимо обеспечить поиск в заголовках HTML-документов. Создадим для этого группу разделов с тегом TITLE, а затем удалим и заново создадим индекс:

```
SQL> begin
  2   ctx_ddl.create_section_group('my_section_group','BASIC_SECTION_GROUP');
  3  ctx_ddl.add_field_section(
  4      group_name => 'my_section_group',
  5       section_name => 'Title',
  6       tag          => 'title',
  7       visible      => FALSE);
  8  end;
  9  /

PL/SQL procedure successfully completed.

SQL> drop index my_html_idx;

Index dropped.

SQL> create index my_html_idx on my_html_docs(html_text)
  2  indextype is ctxsys.context
  3  parameters('section group my_section_group')
  4  /
Index created.
```

Мы создали новую группу разделов, **MY_SECTION_GROUP**, и добавили в нее раздел с именем **Title**. Обратите внимание, что раздел соответствует тегу title и будет неви-

дим. Если раздел помечен как видимый, текст между соответствующими тегами считается частью документа. Если же раздел помечен как невидимый, текст между начальным и конечным тегами рассматривается отдельно от документа и будет доступен только при поиске в соответствующем разделе.

Как и большинство современных языков разметки (например, XML, HTML, WML), начальный тег в interMedia Text начинается символом < и заканчивается символом >. Конечный тег начинается с последовательности символов </ и заканчивается символом >.

```
SQL> select id
  2    from my_html_docs
  3  where contains(html_text, 'Oracle') > 0
  4  /

no rows selected
```

Запрос, прежде возвращавший одну строку, теперь строк не возвращает, а мы ведь всего лишь задали группу разделов для индекса interMedia Text. Вспомните, что раздел Title был задан как невидимый, поэтому текст в тегах **title** рассматривается как подчиненный документ.

```
SQL> select id
  2    from my_html_docs
  3  where contains(html_text, 'Oracle within title') > 0
  4  /

        ID

         1
```

Теперь можно выполнить запрос, выполняющий поиск только в разделах **title** всех документов. Если же попытаться искать текст самого тега, окажется, что компонент interMedia Text тег не проиндексировал:

```
SQL> select id
  2    from my_html_docs
  3  where contains( html_text, 'title' ) > 0
  4  /

no rows selected
```

Хотя ранее я задал собственный тип группы разделов на основе группы разделов BASIC_SECTION_GROUP, в состав компонента interMedia входят также заранее заданные группы разделов со стандартными системными установками для языков HTML и XML (HTML_SECTION_GROUP и XML_SECTION_GROUP). Использование такой группы разделов не определяет автоматически разделы для всех возможных элементов HTML и XML. Это надо делать самому. Однако при использовании указанных групп разделов компонент interMedia Text сможет корректно преобразовать размеченный документ в обычный текст. Попробуем применить соответствующую заданную группу в рассмотренном примере:

```
SQL> drop index my_html_idx;

Index dropped.
```

```
SQL> create index my_html_idx on my_html_docs(html_text)
  2  indextype is ctxsys.context
  3 parameters('section group ctxsys.html_section_group')
  4  /

Index created.

SQL> select id
  2    from my_html_docs
  3   where contains(html_text, 'html') > 0
  4   /

no rows selected
```

Оказывается, задав системные установки, соответствующие группе разделов **HTML_SECTION_GROUP,** мы избежали индексирования строк, являющихся тегами разметки языка HTML. Это не только повышает точность запросов к документам, но и сокращает общий размер индекса interMedia Text. Предположим, необходимо найти слово **title** во всех хранящихся HTML-документах. Если не использовать для индекса interMedia Text группу **HTML_SECTION_GROUP,** в ответ на этот запрос могут быть выданы все HTML-документы, содержащие раздел **title** (речь идет о части HTML-документа, между тегами **<title>** и **</title>).** Игнорируя теги и ограничиваясь исключительно содержимым HTML-документов, можно существенно повысить точность поиска.

Рассмотрение обработки XML-документов начнем с примера. Предположим, необходимо управлять подборкой XML-документов и обеспечить интерфейс для запросов к структурным элементам этих документов. Чтобы усложнить задачу, предположим, что не все собранные XML-документы соответствуют одному и тому же определению структуры, задаваемому *определением типа документа* (Document Type Definition — DTD).

По аналогии с предыдущим примером можно подумать, что необходимо определить все элементы XML-документов, в которых может потребоваться поиск, а затем задать раздел interMedia Text для каждого из этих элементов. К счастью, компонент interMedia Text включает средства автоматического создания и индексирования разделов по имеющимся в документе тегам.

Появившаяся в компоненте interMedia Text версии Oracle 8.1.6, группа разделов **AUTO_SECTION_GROUP** работает аналогично группе разделов XML_SECTION_GROUP, но снимает с разработчика приложений необходимость заранее определять все разделы. Группа разделов **AUTO_SECTION_GROUP** требует от компонента interMedia Text автоматически создать разделы для всех непустых тегов в документе. Хотя разработчик явно может сопоставить тегу любое имя раздела, при такой автоматической генерации имена разделов будут совпадать с соответствующими тегами.

```
SQL> create table my_xml_docs
  2  (id     number primary key,
  3  xmldoc varchar2(4000)
  4  )
  5  /

Table created.

SQL> insert into my_xml_docs(id, xmldoc)
  2  values(1,
```

```
  3  '<appointment type="personal">
  4      <title>Team Meeting</title>
  5      <start_date>31-MAR-200K/start_date>
  6      <start_time>1100</start_time>
  7      <notes>Review projects for QK/notes>
  8      <attendees>
  9          <attendee>Joel</attendee>
 10          <attendee>Tom</attendee>
 11      </attendees>
 12  </appointment>')
 13  /

1 row created.

SQL> commit;

Commit complete.

SQL> create index my_xml_idx on my_xml_docs(xmldoc)
  2  indextype is ctxsys.context
  3  parameters('section group ctxsys.auto_section_group')
  4  /
Index created.
```

Таким образом, без всяких дополнительных действий со стороны разработчика приложения компонент interMedia Text автоматически создал разделы для всех тегов, содержащихся в индексируемых XML-документах. Так что, если необходимо найти документы, содержащие слово **projects** в элементе **note,** достаточно выполнить следующий оператор:

```
SQL> select id
  2    from my_xml_docs
  3   where contains(xmldoc,  'projects within notes') > 0
  4   /

        ID

         1
```

В процессе автоматического создания разделов созданы специальные *разделы зон* (zone section). В предыдущих примерах определения разделов создавались исключительно *разделы полей* (field section). В отличие от разделов полей, разделы зон могут перекрываться и быть вложенными. Поскольку при использовании группы разделов **AUTO_SECTION_GROUP** компонент interMedia Text создает разделы зон для всех непустых тегов, можно выполнять запросы вида:

```
SQL> select id
  2    from my_xml_docs
  3   where contains(xmldoc,  'projects within appointment') > 0
  4   /

        ID

         1
```

```
SQL> select id
  2    from my_xml_docs
  3  where contains(xmldoc, 'Joel within attendees') > 0
  4  /

        ID

         1
```

Раздел, указанный в предыдущих запросах, не содержит искомых терминов явно: они вложены в структурные элементы этого раздела. Использование разделов зон и автоматического создания разделов позволяет управлять областью поиска в XML-документе, расширяя или сужая ее в соответствии с необходимостью.

Используя группы разделов, можно потребовать от компонента interMedia Text проиндексировать атрибуты тегов. При использовании группы разделов **AUTO_SECTION_GROUP** значения атрибутов разделов автоматически выбираются и индексируются.

Итак, если необходимо найти все персональные задания, другими словами, XML-документы, содержащие строку **personal** как значение атрибута **type** тега **appointment,** можно выполнить следующий запрос:

```
SQL> select id
  2    from my_xml_docs
  3  where contains(xmldoc, 'personal within appointinent@type') > 0
  4  /

        ID

         1
```

Как видите, задание и индексирование разделов — очень мощное средство компонента interMedia Text. Следует, однако, помнить, что группу **AUTO_SECTION_GROUP** можно использовать **не всегда.** Хотя и есть возможность потребовать от компонента interMedia Text при автоматическом создании разделов не индексировать определенные теги, в конечном итоге в разделы может быть выделено и проиндексировано слишком много элементов документа, что "загрязняет" индекс. Общий размер индекса чрезвычайно увеличивается, что может резко снизить производительность поиска. Автоматическое создание разделов — мощное средство, но его надо использовать осторожно.

Проблемы

При использовании компонента interMedia Text следует учитывать возможность возникновения ряда проблем. Не все они очевидны или возникают достаточно часто, поэтому я опишу наиболее типичные, с которыми мне приходилось сталкиваться.

Компонент interMedia Text — это НЕ система документооборота

Это не проблема, скорее следствие неверного представления о назначении компонента interMedia Text. Я слышал, как при упоминании компонента interMedia Text клиенты и сотрудники корпорации Oracle называли его *системой документооборота* (document

management solution). Без сомнения, interMedia Text **не является** системой документооборота.

Документооборот — отдельная полноценная наука. Система документооборота предлагает набор средств, поддерживающих весь жизненный цикл документов. Она должна обеспечивать регистрацию входящих и исходящих документов, их логическую структуризацию, хранение нескольких версий документа и списков контроля доступа, интерфейс для поиска текста, а также возможность публикации документов.

Компонент interMedia Text , конечно, не является системой документооборота. Он может использоваться в полной системе документооборота и обеспечивать многие из необходимых функций. Корпорация Oracle тесно интегрировала средства interMedia Text в состав системы Oracle Internet File System, которую можно использовать для управления содержимым сайтов (content management), и которая обеспечивает базовые функции системы документооборота.

Синхронизация индекса

Часто необходимо создать систему на базе средств interMedia Text и выполнять в фоновом режиме процесс ctxsrv для периодической синхронизации индекса interMedia Text, причем синхронизация должна происходить в реальном времени. Одна из проблем, которые могут возникнуть при многократной и частой синхронизации индекса interMedia Text для большого набора документов, связана с его разрастанием и фрагментацией.

Нет простого правила, позволяющего определить, когда следует периодически синхронизировать индексы большими пакетами, а когда лучше синхронизировать их с помощью процесса ctxsrv сразу после фиксации изменений в документах. Во многом это зависит от сути приложения, частоты изменения текста документов, общего количества и размера документов.

В качестве примера можно привести мой Web-сайт AskTom. На этом сайте пользователи Oracle задают технические вопросы о программных продуктах Oracle. Вопросы просматриваются, на них даются подробные (я надеюсь) ответы, и эти ответы публикуются. Опубликованные вопросы и ответы вставляются в таблицу, проиндексированную с помощью индекса interMedia Text. На Web-сайте AskTom есть страница поиска по этой таблице опубликованных вопросов и ответов.

Общее количество строк в этой системе сравнительно невелико (менее 10000 на момент написания этой главы). Изменения в системе практически никогда не происходят; после публикации вопросов и ответов они почти никогда не изменяются и не удаляются. Каждый день вставляется обычно не более 25 строк, и вставки эти выполняются в течение всего дня. Мы выполняем синхронизацию индекса с помощью процесса ctxsrv, работающего в фоновом режиме, что идеально подходит для данной системы, предназначенной прежде всего для поиска в условиях небольшого количества вставок и изменений.

Если же предполагается загрузка в таблицу по миллиону документов в неделю, не стоит индексировать их с помощью процесса **ctxsrv.** В этом случае имеет смысл синхронизировать индекс interMedia Text пакетами максимально большого размера, при котором еще не требуется откачка страниц памяти на диск. Выстроив запросы индексирова-

ния в очередь и выполняя их большими пакетами, можно получить в результате более компактный индекс.

Независимо от выбранного метода синхронизации следует периодически оптимизировать индексы interMedia Text с помощью оператора **ALTER INDEX REBUILD.** Процесс оптимизации позволит получить не только более компактный индекс, но и очистит его от информации, оставшейся после прежних логических удалений.

Индексирование информации вне базы данных

Компонент interMedia Text не требует помещать текстовые данные в базу данных. С его помощью можно индексировать данные, содержащиеся в документах в файловой системе сервера или даже доступных извне по адресу URL.

Когда данные находятся в базе данных Oracle, все изменения в них автоматически обрабатываются компонентом interMedia Text. Когда же источник данных находится вне базы данных, синхронизацию индекса с изменившимися внешними данными должен обеспечить разработчик приложения.

Изменить отдельную строку проще всего, изменив один из столбцов, по которому создан индекс interMedia Text. Например, если бы я использовал следующую таблицу и индекс для поддержки списка проиндексированных адресов URL:

```
SQL> create table my_urls
  2  (id number primary key,
  3   theurl varchar2(4000)
  4   )
/

Table created.

SQL> create index my_url_idx on my_urls(theurl)
  2  indextype is ctxsys.context
  3 parameters('datastore ctxsys.url_datastore')
  4  /

Index created.
```

Я мог бы "обновить" индекс для конкретной строки, выполнив:

```
SQL> update my_urls
  2     set theurl = theurl
  3   where id = 1
  4  /

0 rows updated.
```

Службы обработки документов

Под "службами обработки документов" я подразумеваю набор средств компонента interMedia Text для преобразования документа в текстовый вид или формат HTML, с возможным выделением слов, найденных в документе в результате поиска.

Многие ошибочно считают, что компонент interMedia Text сохраняет всю необходимую информацию для полного воссоздания документа. Это ошибочное представление приводит к выводу, что после индексирования исходный текст документа можно уда-

лить. Это верно, если придется только выполнять запросы к проиндексированной информации, но не верно, если предполагается поддержка тех или иных служб обработки документов.

Например, если создать индекс interMedia Text с помощью **URL_DATASTORE** и попытаться сгенерировать HTML-представление найденной строки с помощью процедуры **CTX_DOC.FILTER,** ее вызов завершится сообщением об ошибке, если документ с соответствующим адресом URL недоступен. Компоненту interMedia Text для выполнения этого действия необходим доступ к исходному документу. Это относится также к файлам, хранящимся в файловой системе вне базы данных и проиндексированным как источник данных **FILE_DATASTORE.**

Индекс-каталог

Во многих ситуациях индекс interMedia Text предоставляет намного больше возможностей, чем требуется приложению. Использование индекса interMedia Text, кроме того, требует выполнения определенных действий по сопровождению, обеспечивающих его оптимизацию, синхронизацию и т.д. Для поддержки приложений, которым не нужны все функциональные возможности индекса interMedia Text, в версии компонента interMedia Text в СУБД Oracle 8.1.7 появился новый тип индекса — индекс-каталог, или, сокращенно, ctxcat.

Обычно большая часть текстовых данных не хранится в базе данных в виде большого набора документов. Во многих приложениях баз данных текст обычно неформатирован, состоит из небольших фрагментов и не разбит на логические разделы, а размер текстовых фрагментов настолько мал, что их качественный лингвистический анализ невозможен. Кроме того, такого рода приложения баз данных часто запрашивают текстовые данные по условиям на другие столбцы., Рассмотрим, например, базу данных, в которой регистрируются поступившие сообщения о проблемах и предлагаемые способы решения этих проблем. В соответствующей таблице может использоваться, скажем, 80-символьное поле произвольного текста — тема (суть проблемы) и большое текстовое поле с описанием проблемы и способов ее решения. Кроме того, в таблице могут быть и другие столбцы со структурированной информацией, например датой поступления сообщения о проблеме, кодом аналитика, который над ней работает, кодом программного продукта, с которым связана проблема, и т.п. Мы имеем сочетание текста (по определению не являющегося документом) и структурированных данных. К этой таблице будут часто выполняться запросы типа "найти все проблемы, связанные с СУБД (программный продукт) версии 8.1.6 (еще один атрибут), где упоминается ошибка ORA-01555 в поле темы (текстовый поиск)". Именно для таких приложений и был создан индекс-каталог.

Как и можно было ожидать от "урезанной" версии полного индекса, индекс **ctxcat** имеет ряд ограничений. Поддерживаемые этим индексом операторы запросов являются подмножеством операторов "полного" индекса interMedia Text. Данные, которые индексируются с помощью индекса **ctxcat,** должны содержаться в базе данных Oracle в текстовом виде. Более того, индекс **ctxcat** не допускает использование нескольких языков в одном индексе. Однако даже с учетом этих ограничений индекс-каталог обеспечивает отличную производительность для многих приложений.

Одна из приятных особенностей индекса-каталога — его не надо поддерживать. Операторы ЯМД применяются к этому индексу в рамках транзакции. Поэтому не нужно периодически синхронизировать индекс или запускать процесс **ctxsrv** для синхронизации в фоновом режиме.

Еще одна серьезная причина использовать индекс-каталог связанна с присущей ему поддержкой структурированных запросов. Разработчик приложений может создавать наборы индексирования (index sets), которые позволяют с помощью индекса-каталога эффективно поддерживать как текстовый поиск, так и запросы к другим структурированным данным. Набор индексирования позволяет компоненту interMedia сохранять в индексе структурированную реляционную информацию вместе с индексируемым текстом. Это позволяет компоненту interMedia одновременно использовать поиск текста и поиск структурированной информации для нахождения документов, удовлетворяющих специфическим критериям. Рассмотрим небольшой пример:

```
SQL> create table mynews
  2  (id number primary key,
  3   date_created date,
  4  news_text varchar2(4000))
  5  /

Table created.

SQL> insert into mynews
  2 values(1, '01-JAN-1990', 'Oracle is doing well')
  3  /

1 row created.

SQL> insert into mynews
  2 values(2, '01-JAN-2001', 'I am looking forward to 9i')
  3  /

1 row created.

SQL> commit;

Commit complete.

SQL> begin
  2    ctx_ddl.create_index_set('news_index_set');
  3      ctx_ddl.add_index('news_index_set',  'date_created');
  4 end;
  5  /
SQL> create index news_idx on mynews(news_text)
  2  indextype is ctxsys.ctxcat
  3 parameters('index set news_index_set')
  4  /

Index created.
```

Обратите внимание, что для создания индекса-каталога указан тип индекса **CTXSYS.CTXCAT**. Кроме того, я создал набор индексирования **NEWS_INDEX_SET** и добавил в него столбец **DATE_CREATED**. Это позволит компоненту interMedia Text эффективно обрабатывать запросы, содержащие условия для обоих столбцов: **NEWS_TEXT** и **DATE_CREATED**.

```
SQL> select id
  2    from mynews
  3   where catsearch(news_text, 'Oracle', null) > 0
  4     and date_created < sysdate
  5  /

        ID

         1

SQL> select id
  2    from mynews
  3   where catsearch(news_text, 'Oracle', 'date_created < sysdate') > 0
  4  /

        ID

         1
```

Здесь мы видим оба метода поиска строк, содержащих слово Oracle в тексте сообщения, дата внесения которого, **DATE_CREATED**, предшествует текущему дню. Первый запрос сначала использует interMedia для поиска всех строк, которые могут удовлетворять запросу, а затем просматривает их в поисках тех, у которых значение **DATE_CREATED** меньше, чем **SYSDATE**. Второй, более эффективный запрос, будет использовать индекс interMedia, включающий столбец **DATE_CREATED**, для поиска только тех строк, которые одновременно удовлетворяют критерию поиска текста и условию **DATE_CREATED** < **SYSDATE**. При поиске по индексу-каталогу вместо оператора **CONTAINS** используется оператор **CATSEARCH**. Поскольку ранее был создан набор индексирования, содержащий столбец **DATE_CREATED**, стало возможным задать структурированное условие запроса непосредственно в операторе **CATSEARCH**. С помощью этой информации сервер Oracle может очень эффективно проверить оба условия запроса.

На условия, которые можно указывать в операторе **CATSEARCH**, налагается ряд ограничений, в частности поддерживаются только логические операторы **AND**, **OR** и **NOT**, но для множества приложений этот индекс не только подходит, но и является оптимальным.

Возможные ошибки

Общаясь с разработчиками приложений и клиентами, использующими компонент interMedia Text, я постоянно слышу об одном и том же небольшом наборе проблем. Эти проблемы рассматриваются в данном разделе.

Устаревший индекс

Часто ко мне обращались с вопросом, почему часть информации проиндексирована, но недавно добавленные строки индексом не учитываются. Чаще всего причина этого в том, что индекс interMedia Text не синхронизирован. Выполнив запрос к представлению CTX_USER_PENDING, легко определить, есть ли отложенные изменения, ожи-

дающие индексирования. Если есть, синхронизируйте индекс с помощью одного из описанных ранее методов.

Еще одна типичная причина отсутствия информации в индексе связана с ошибками, особенно в процессе фильтрования. При попытке индексирования документов с помощью фильтров Inso, если формат документа не поддерживается, возникают ошибки, и соответствующий документ не индексируется. В представлении CTX_USER_INDEX_ERRORS можно найти достаточно информации, чтобы выяснить причины возникновения проблем при индексировании.

Ошибки внешней процедуры

В версии компонента interMedia Text, поставлявшейся в составе СУБД Oracle 8.1.5 и 8.1.6, фильтрование текста выполнялось с помощью фильтров Inso, подключаемых через внешние процедуры. Внешние процедуры — это написанные на языке C функции, хранящиеся в разделяемой библиотеке и вызываемые из PL/SQL. Эти внешние процедуры работают в отдельном адресном пространстве, а не в адресном пространстве сервера Oracle.

Если необходимо фильтровать документы с помощью компонента interMedia Text, но поддержка внешних процедур не сконфигурирована на сервере надлежащим образом, возможно получение одного или нескольких сообщений об ошибках*:

```
ORA-29855: error occurred in the execution of ODCIINDEXCREATE routine
ORA-29855: возникла ошибка при выполнении программы ODCIINDEXCREATE

ORA-20000: interMedia text error:
DRG-50704: Net8 listener is not running or cannot start external
procedures

ORA-28575: unable to open RPC connection to external procedure agent
ORA-28575: невозможно открыть соединение RPC с агентом внешней процедуры
```

Исчерпывающую информацию о конфигурировании сервера для поддержки внешних процедур можно найти в руководстве *Net8 Administrator's Guide,* но вот очень простой и быстрый способ проверить, работают ли внешние процедуры. Организовав с сервера базы данных (или из окна командной строки, если сервер работает на платформе Microsoft Windows) сеанс **telnet,** выполните команду **tnsping extproc_connection_data.** Если в результате вы не получите ответ OK, как показано ниже:

```
oracle8i@cmh:/> tnsping extproc_connection_data

TNS Ping Utility for Solaris: Version 8.1.7.0.0 - Production on 30-MAR-
2001 13:46:59

(c) Copyright 1997 Oracle Corporation. All rights reserved.

Attempting to contact (ADDRESS=(PROTOCOL=IPC)(KEY=EXTPROC))
OK (100 msec)
```

значит, поддержка внешних процедур сконфигурирована неправильно.

**** Представлены также тексты сообщений об ошибках, выдаваемые на русском языке сервером Oracle 8.1.6.0.0. - Прим. научн. ред.***

Внешние процедуры не нужны для фильтрования с помощью interMedia Text в версии Oracle 8.1.7. Процесс фильтрования теперь поддерживается самим сервером.

Дальнейшее развитие

С выходом Oracle 9i название компонента interMedia Text снова изменилось — теперь соответствующий компонент называется Oracle Text. Все функциональные возможности, которые были в версиях interMedia Text для Oracle 8i, будут поддерживаться и в Oracle Text, но в версию Oracle 9i добавлен ряд новых полезных возможностей.

Одной из наиболее желательных возможностей для компонента interMedia Text была автоматическая классификация документов по содержанию. Компонент interMedia Text давал разработчику приложений все необходимые средства для тематического анализа и создания резюме документов, на основе которых можно было их классифицировать. В Oracle Text возможность классификации документов просто встроена. Она помогает создавать системы, позволяющие определить, каким запросам будет соответствовать документ.

В компоненте Oracle Text также расширена встроенная поддержка XML-документов. Кроме собственно содержимого, в XML-документ входят структурированные метаданные. Между элементами XML-документа имеются неявные взаимосвязи, и для выражения этих взаимосвязей можно использовать структурированные метаданные. Спецификация **XPath,** рекомендованная консорциумом W3C **(http://www.w3.org/TR/xpath),** предлагает способ получения элементов XML-документа на основе содержимого и относительной структуры этих элементов. Компонент Oracle Text включает новый оператор XPATH, позволяющий создавать SQL-запросы на основе структурных элементов документа.

Это лишь некоторые из новых возможностей, появившихся в компоненте interMedia Text/Oracle Text с выходом сервера версии Oracle 9i.

Резюме

В этой главе мы рассмотрели богатый набор функциональных возможностей компонента interMedia Text, а также способы их использования в разнообразных приложениях. Хотя в этой главе описаны многие особенности и средства interMedia Text, гораздо больше осталось за рамками обсуждения. Можно использовать тезаурус, задавать специализированный лексический анализатор, генерировать HTML-представление для всех документов, независимо от их исходного формата, сохранять запросы для дальнейшего использования и даже создавать собственные списки стоп-слов.

Компонент interMedia Text — обширная тема для обсуждения, его невозможно описать в одной главе. Наряду с богатством возможностей компонент interMedia Text отличается простой использования и понятностью. После прочтения этой главы у вас должно сложиться четкое понимание того, как реализован компонент interMedia Text и как его использовать в приложениях.

18

Внешние процедуры на языке C

Вначале в качестве языка программирования сервера Oracle и клиентских приложений, **работающих на самом сервере** Oracle, использовался исключительно PL/SQL. В версии Oracle 8.0 появилась возможность создавать хранимые процедуры на других языках. Эта возможность — поддержка *внешних процедур* — распространяется на хранимые процедуры на языке C (и все процедуры, которые можно вызвать из языка C) и на языке Java. В этой главе мы сконцентрируемся исключительно на языке C, а следующая глава посвящена использованию Java.

В этой главе внешние процедуры рассматриваются с точки зрения архитектуры и показано, как они были реализованы разработчиками ядра Oracle. Кроме того, мы разберемся, как сконфигурировать сервер для поддержки внешних процедур и как должен конфигурироваться сервер с учетом защиты. Я продемонстрирую, как написать внешнюю процедуру с помощью прекомпилятора Oracle Pro*C. Эта внешняя процедура будет использоваться для записи содержимого любого большого объекта в файловую систему сервера. Однако прежде чем перейти к этому примеру, рассмотрим базовый пример, демонстрирующий, как передавать значения основных типов данных из языка PL/SQL в функции на языке C и получать результаты. Этот базовый пример позволит также создать шаблон для быстрой разработки всех последующих внешних процедур на языке C. Вы узнаете, как создавать код на языке C с учетом возможности использования его для выполнения сервером Oracle. Мы также рассмотрим реализацию SQL-оболочки для внешней процедуры. Я расскажу, как обеспечить возможность ее вызова из языков SQL и PL/SQL и как программировать оболочку, чтобы ее легко было использовать тем, кому нужен соответствующий код. Наконец, мы рассмотрим преимущества и недостатки вне-

шних процедур, а также различные ошибки сервера (ошибки **ORA-XXXX),** которые могут возникнуть при их использовании.

В примере на языке Pro*C будет создаваться недостающее средство сервера. В составе сервера Oracle поставляется пакет **DBMS_LOB** для работы с большими объектами. В этом пакете есть процедура **loadfromfile,** позволяющая читать в большой объект базы данных содержимое любого файла в файловой системе сервера. Однако в этом пакете нет функции **writetofile,** которая могла бы записывать в файловую систему содержимое большого объекта, а такая потребность часто возникает. Мы решим эту задачу, создав собственный пакет **LOB_IO.** Этот пакет позволит записывать любой большой объект типа **BLOB, CLOB** или **BFILE** в отдельный файл вне базы данных (так что для объектов типа **BFILE** мы по сути создадим команду копирования, поскольку исходный объект **BFILE** уже находится вне базы данных).

Когда используются внешние процедуры?

Один язык или одна среда не может обеспечить средства и функции на все случаи жизни. У каждого языка есть недостатки: не все можно сделать с его помощью, недостает возможностей или средств, о которых разработчики не подумали. При разработке программ на языке C мне иногда приходится программировать на ассемблере. При программировании приложений для Oracle на Java иногда удобно использовать код на языке PL/SQL. Суть в том, что не нужно всегда использовать язык "низкого уровня" — иногда необходимо переходить на более высокий уровень. Внешние процедуры можно считать переходом на более "низкоуровневый" язык. Они обычно используются для интеграции существующего кода на языке C в виде библиотек функций (например, DLL — динамически компонуемых библиотек в Windows, созданных сторонним производителем, которые необходимо вызывать с сервера) или для расширения функциональных возможностей существующих пакетов, как в нашем случае. Именно эту технологию используют разработчики сервера Oracle для расширения его возможностей. Например, мы уже рассмотрели, как эта возможность использовалась в компоненте interMedia (в предыдущей главе) и в пакете **DBMS_OLAP** (в главе 13, посвященной материализованным представлениям).

Первая хранимая процедура, которую я написал, представляла собой реализацию простого клиента TCP/IP. С ее помощью в версии сервера 8.0.3 я получил возможность в PL/SQL открывать сокет TCP/IP для подключения к существующему серверу и обмена сообщениями с ним. Я мог подключаться к серверу дискуссионных групп по протоколу Net News Transport Protocol (NNTP), к серверу электронной почты по протоколам Internet Message Access Protocol (IMAP), Simple Mail Transfer Protocol (SMTP) или Post Office Protocol (POP), к Web-серверу и т.д. "Научив" язык PL/SQL использовать сокеты, я открыл широкий спектр новых возможностей. Теперь я мог:

- посылать сообщение электронной почты из триггера с помощью протокола SMTP;
- включать сообщения электронной почты в базу данных с помощью протокола POP;

- индексировать сообщения дискуссионных групп с помощью компонента interMedia Text и протокола NNTP;
- обращаться к любой доступной сетевой службе.

Я стал использовать сервер Oracle также в качестве клиента прочих серверов. После включения полученных от них данных в свою базу я мог выполнять с ними множество действий (индексировать, выполнять поиск, представлять в другом виде и т.д.).

Со временем это средство начали использовать настолько часто, что теперь оно стало интегрированной возможностью сервера. Начиная с версии 8.1.6 сервера Oracle, все возможности, которые обеспечивал простой клиент TCP/IP, теперь реализуются в пакете **UTL_TCP**.

С тех пор я написал еще несколько внешних процедур. Одни — для получения времени с помощью системных часов с большей точностью, чем возвращает встроенная функция **SYSDATE**, другие — для выполнения команд операционной системы, определения часового пояса системы или для получения списка файлов в указанном каталоге. Последней мной написана функция для записи во внешний файл содержимого любого большого объекта: символьного (Character **LOB — CLOB**), двоичного (Binary LOB — **BLOB**) или хранящегося во внешнем файле (**BFILE**). Полезным побочным эффектом созданного при этом пакета является обеспечение для двоичных файлов возможностей, предоставляемых пакетом **UTL_FILE** (пакет **UTL_FILE** не позволяет создавать двоичные файлы). Поскольку сервер поддерживает *временные большие объекты* (Temporary LOB) и обеспечивает возможность записи (**WRITE**) во временный большой объект, этот новый пакет, который мы собираемся реализовать, даст возможность записывать из PL/SQL любой двоичный файл. Итак, этот пакет позволит:

- экспортировать любой большой объект во внешний файл на сервере;
- записывать на сервере двоичный файл практически любого размера и с любыми данными (аналогично пакету **UTL_FILE**, который работает с текстовыми данными, но работу с произвольными двоичными данными не поддерживает).

Из всего сказанного ясно, что причины использования внешних процедур могут быть многочисленны и разнообразны. Обычно они используются для:

- реализации отсутствующих функциональных возможностей;
- интегрирования существующего кода, который проверяет корректность данных;
- ускорения обработки; скомпилированный код на языке C всегда будет выполнять операции, требующие большого объема вычислений, быстрее, чем при реализации их на интерпретируемых языках PL/SQL или Java.

Как обычно, решение использовать что-то вроде хранимых процедур требует определенных издержек. Дополнительные издержки связаны с разработкой кода на языке C, что, как мне кажется, сложнее, чем разработка на PL/SQL. Приходится также жертвовать переносимостью или даже идти на потенциальную невозможность переноса кода. Если разработана DLL-библиотека для Windows, нет гарантии, что написанный исходный код можно будет использовать на UNIX-машине, и наоборот. Я считаю, что внешнюю процедуру надо использовать лишь тогда, когда невозможно решить задачу с помощью языка PL/SQL.

Как реализована поддержка внешних процедур?

Внешние процедуры выполняются процессом, физически отделенным от процессов сервера. Это сделано из соображений защиты. Хотя технически можно динамически загружать DLL-библиотеку (в Windows) или файл **.so** (Shared Object code — разделяемый объектный код, скажем, в Solaris) во время выполнения и в существующих процессах сервера, при этом сервер будет подвергаться неоправданному риску. Код библиотеки будет иметь доступ к тому же пространству памяти, что и процессы сервера, в том числе и к системной глобальной области Oracle (SGA). В результате этот посторонний код может случайно повредить базовые структуры данных СУБД, что приведет к потере данных или сбою экземпляра. Чтобы избежать этого, внешние процедуры выполняются отдельными процессами, не использующими разделяемые области памяти сервера.

В большинстве случаев отдельный процесс будет работать от имени пользователя, не **являющегося владельцем программного обеспечения Oracle.** Причина та же, что и в случае выполнения внешних процедур отдельным процессом — безопасность. Пусть, например, мы собираемся создать внешнюю процедуру, которая сможет записывать файлы на диск (как это делает процедура, рассматриваемая далее). Предположим, сервер работает в среде UNIX, и внешняя процедура выполняется от имени владельца программного обеспечения Oracle. Пользователь вызывает новую функцию и "просит" ее записать объект **BLOB** в файл **/d0l/oracle/data/system.dbf.** Поскольку этот код выполняется от имени пользователя — владельца программного обеспечения Oracle, этот вызов сработает и непреднамеренно запишет содержимое какого-то большого двоичного объекта вместо системного табличного пространства. Мы можем этого даже и не заметить до момента остановки и перезапуска сервера (много дней спустя). Если бы внешняя процедура выполнялась от имени менее привилегированного пользователя, это не могло бы случиться (этот пользователь не имел бы права на запись файла **system.dbf)** Поэтому в разделе, посвященном конфигурированию сервера для поддержки внешних процедур, мы рассмотрим, как настроить безопасный процесс прослушивания **EXTPROC** (EXTemal PROCedure), работающий от имени другой учетной записи ОС. Причины такой настройки примерно те же, что и в случае запуска Web-серверов от имени пользователя nobody в UNIX или от имени учетной записи с минимальными привилегиями в Windows.

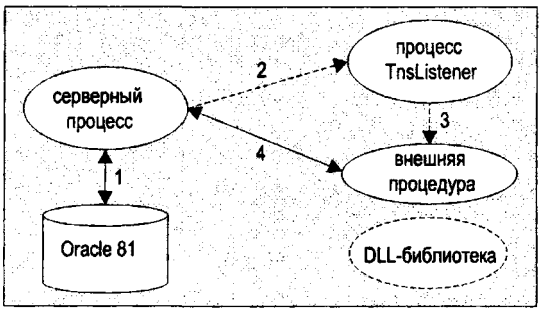
Итак, при вызове внешней процедуры сервер Oracle будет автоматически создавать процесс ОС под названием **EXTPROC.** Для этого он связывается с *процессом прослушивания* Net8 (Net8 listener). Процесс прослушивания Net8 будет автоматически создавать процесс **EXTPROC** точно так же, как он порождает выделенные или разделяемые серверы. В среде Windows NT это можно увидеть с помощью утилиты **tlist** из набора NT Resource Toolkit, выдающей дерево процессов и подпроцессов. Например, я запустил сеанс, из которого обратился к внешней процедуре, а затем выполнил команду **tlist -t** и получил следующее:

```
C:\bin>tlist -t
System Process (0)
System (8)
  smss.exe (140)
    csrss.exe (164)
```

```
winlogon.exe (160)
  services.exe (212)
    svchost.exe (384)
    SPOOLSV.EXE (412)
    svchost.exe (444)
    regsvc.exe (512)
    stisvc.exe (600)
    ORACLE.EXE (1024)
      ORADIM.EXE (1264)
    TNSLSNR.EXE (1188)
      EXTPROC.EXE (972)
  lsass.exe (224)
```

Это показывает, что процесс **TNSLSNR.EXE** является родительским для процесса **EXTPROC.EXE**. Процесс **EXTPROC** и процесс сервера теперь могут взаимодействовать. Что еще важнее, процесс **EXTPROC** может динамически загружать пользовательские DLL-библиотеки (или файлы .so/.sl/.a в ОС UNIX).

Архитектурно это выглядит следующим образом:



Происходит следующее.

1. Пользовать подключается к СУБД. При этом либо запускается процесс выделенного сервера, либо используется один из разделяемых серверных процессов.
2. Пользователь вызывает внешнюю процедуру. Поскольку это первый вызов, серверный процесс связывается с процессом **TNSLISTENER** (процессом прослушивания Net8).
3. Процесс прослушивания Net8 запускает (или находит в пуле запущенных свободный) процесс выполнения внешних процедур для сеанса. Этот процесс загружает запрошенную DLL-библиотеку (или файл **.so/.sl/.a** в ОС **UNIX).**
4. Теперь можно взаимодействовать с процессом выполнения внешних процедур, который будет обеспечивать обмен данными между языками SQL и C.

Конфигурирование сервера

Сейчас я опишу настройку реквизитов, которую необходимо провести, чтобы обеспечить выполнение внешних процедур. Для этого придется настраивать файлы

LISTENER.ORA и **TNSNAMES.ORA** на сервере, а не на клиентской машине. После полной установки эти файлы должны быть автоматически сконфигурированы для поддержки служб внешних процедур (**EXTPROC**). В этом случае конфигурационный файл **LISTENER.ORA** будет иметь примерно такой вид:

```
# LISTENER.ORA. Network Configuration File:
#             C:\oracle\network\admin\LISTENER.ORA
# Generated by Oracle configuration tools.

LISTENER =
   (DESCRIPTION_LIST =
     (DESCRIPTION =
       (ADDRESS_LIST =
         (ADDRESS = (PROTOCOL = TCP)(HOST = tkyte-del)(TORT = 1521))
       )
       (ADDRESS_LIST =
         (ADDRESS = (PROTOCOL = IPC)(KEY = EXTPROC1))
       )
     )
   )

SID_LIST_LISTENER =
   (SID_LIST =
     (SID_DESC -
       (SID_NAME = PLSExtProc)
       (ORACLE_HOME = C:\oracle)
       (PROGRAM = extproc)
     )
     (SID_DESC =
       (GLOBAL_DBNAME = tkyte816)
       (ORACLE_HOME = C:\oracle)
       (SID_NAME = tkyte816)
     )
   )
```

Следующие установки в файле процесса прослушивания существенны для использования внешних процедур.

- **(ADDRESS = (PROTOCOL = IPC)(KEY = EXTPROC1)).** Задает **IPC**-адрес. Запомните значение **KEY.** Это значение может быть произвольным, просто запомните его. В некоторых системах регистр символов в значении **KEY** существенен, что тоже необходимо учитывать.
- **(SID_DESC = (SID_NAME = PLSExtProc,).** Запомните значение **SID_NAME, PLSExtProc** или что-то подобное. По умолчанию это значение **SID** будет равно **PLSExtProc.**

Файл **LISTENER.ORA** можно сконфигурировать и вручную с помощью простого текстового редактора или с помощью программы Net8 Assistant. Настоятельно рекомендуется использовать программу Net8 Assistant, поскольку минимальная ошибка в конфигурационном файле, например незакрытая круглая скобка, сделает его бесполезным. При использовании Net8 Assistant следуйте процедуре, описанной в системе оператив-

ной справки в разделе **NetAssistantHelp, Local, Listeners, How To..., и Configure External Procedures for the Listener.**

После изменения файла **LISTENER.ORA** не забудьте остановить и запустить процесс прослушивания с помощью команд **lsnrctl stop и lsnrctl start** из командной строки.

Следующий конфигурационный файл — **TNSNAMES.ORA**. Этот файл должен находиться в каталоге, который сервер будет использовать при разрешении имен. Обычно файл **TNSNAMES.ORA** находится на клиенте и используется для поиска сервера. В данном случае сервер сам должен найти службу по имени. Файл **TNSNAMES.ORA** будет иметь примерно такой вид:

```
# TNSNAMES.ORA Network Configuration
#                File:C:\oracle\network\admin\TNSNAMES.ORA
# Generated by Oracle configuration tools.

EXTPROC_CONNECTION_DATA =
  (DESCRIPTION =
    (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL = IPC)(KEY = EXTPROC1))
    )
    (CONNECT_DATA =
      (SID = PLSExtProc)
      (PRESENTATION = RO)
    )
  )
```

В этом конфигурационном файле существенно следующее.

- **EXTPROC_CONNECTION_DATA**. Имя службы, которую будет искать сервер. Это имя использовать обязательно. Далее будет рассмотрена проблема, связанная с установкой параметра names.default_domain в конфигурационном файле SQLNET.ORA.
- **(ADDRESS = (PROTOCOL = IPC)(KEY = EXTPROC1)).** Этот параметр должен быть таким же, как в файле **LISTENER.ORA**. В частности, значения компонента **KEY** = должны совпадать.
- **(CONNECT_DATA =(SID = PLSExtProc)**. Значение после **SID =** должно соответствовать значению **SID** в конструкции **(SID_DESC = (SID_NAME = PLSExtProc)** в файле **LISTENER.ORA**.

С именем EXTPROC_CONNECTION_DATA связана следующая проблема. Если в файле SQLNET.ORA задан стандартный домен, он должен быть включен в запись TNSNAMES. Так что, если в файле SQLNET.ORA есть следующая установка:

```
names.default_domain = world
```

необходимо задать в файле **TNSNAMES.ORA** значение **EXTPROC_CONNECTION_DATA.world**, а не просто **EXTPROC_CONNECTION_DATA**.

Любые ошибки в представленных выше конфигурационных файлах почти наверняка приведут к выдаче сообщения об ошибке ORA-28575, представленного ниже:

```
declare
```

```
ERROR at line 1:
ORA-28575: unable to open RFC connection to external procedure agent
ORA-06512: at "USERNAME.PROCEDURE_NAME", line 0
ORA-06512: at line 5
```

При получении этого сообщения об ошибке имеет смысл проверить следующее:

- доступна ли программа **extproc** и является ли она выполняемой;
- правильно ли настроена среда сервера для использования внешних процедур;
- правильно ли сконфигурирован процесс прослушивания.

Сейчас мы детально рассмотрим каждый из этих шагов. Предполагается, что вы уже получили сообщение об ошибке **ORA-28575** по какой-то причине.

Проверка программы extproc

Если после конфигурирования поддержки внешних процедур вы получаете сообщение об ошибке **ORA-28575,** прежде всего необходимо проверить наличие программы extproc и возможность ее выполнения. Это легко сделать из командной строки — как в Windows, так и в UNIX. Это надо делать, зарегистрировавшись как пользователь, от имени которого будет **запускаться процесс прослушивания** (поскольку именно этот процесс и будет выполнять программу **extproc),** чтобы убедиться, что этот пользователь имеет все необходимые права. Затем нужно выполнить следующую команду:

```
C:\oracle\BIN>.\extproc.exe

Oracle Corporation———SATURDAY  AUG 05 2000 14:57:19.851
Heterogeneous Agent based on the following module(s):
   — External Procedure Module

C:\oracle\BIN>
```

Результат работы должен быть подобен представленному выше. Обратите внимание, что я выполнял команду из каталога **[ORACLE_HOME]\bin,** поскольку именно в нем и находится программа **extproc.exe.** Если выполнить эту программу не удается, значит, установка выполнена некорректно или необходимо исправить конфигурацию на уровне операционной системы.

Проверка среды сервера

В среде сервера нужно проверить несколько установок. Прежде всего проверьте, используется ли **соответствующий** файл **TNSNAMES.ORA** и правильно ли он сконфигурирован. Например, на моей UNIX-машине с помощью программы **truss** я получил следующую информацию:

```
$ setenv TNS_ADMIN /tmp

$ truss sqlplus /@ora8i.us.oracle.com |& grep TNSNAMES

access("/export/home/tkyte/.TNSNAMES.ORA", 0)  Err#2 ENOENT
access("/tmp/TNSNAMES.ORA", 0)                 Err#2 ENOENT
```

```
access(/var/opt/oracle/TNSNAMES.ORA", 0)       Err#2 ENOENT
access(Vexport/home/oracle8i/network/admin/TNSNAMES.ORA", 0) = 0
```

Итак, сервер Oracle искал файл **TNSNAMES.ORA:**

- в моем начальном каталоге;
- в каталоге, задаваемом переменной среды **TNS_ADMIN;**
- в каталоге **/var/opt/oracle;**
- и, наконец, в каталоге **$ORACLE_HOME/network/admin/.**

Типичная ошибка: файл **TNSNAMES.ORA** конфигурируется в каталоге **[ORACLE_HOME]/network/admin, но** установлена переменная среды **TNS_ADMIN,** которая заставляет сервер Oracle искать файл **TNSNAMES.ORA в** другом месте. Поэтому проверьте, используется ли соответствующий файл **TNSNAMES.ORA** (чтобы добиться однозначности, просто установите переменной среды **TNS_ADMIN** требуемое значение **перед запуском** сервера; это гарантирует, что сервер Oracle использует именно тот экземпляр файла, который **необходимо).**

Убедившись, что используется соответствующий файл **TNSNAMES.ORA,** надо поискать ошибки конфигурирования в этом файле. Следует проверить в соответствии с представленными выше примерами, настроены ли компоненты **(ADDRESS = (PROTOCOL = IPC)(KEY = EXTPROC1)) и (CONNECT_DATA = (SID = PLSExtProc))** корректно. Сравните их с содержимым файла **LISTENER.ORA.** Если для установки использовалась программа Net8 Assistant, заботиться о соответствии круглых скобок не придется. Если файл редактируется вручную, будьте очень внимательны. Одна незакрытая или стоящая не на своем месте круглая скобка не позволит использовать соответствующуюзапись.

Проверив корректность этих компонентов, обратите внимание на имя записи в файле **TNSNAMES.ORA.** Это должна быть запись **EXTPROC_CONNECTION_DATA.** Никакое другое имя не подходит (к имени записи может быть только добавлено **имя** домена). Проверьте правильность написания имени. Одновременно сверьтесь с конфигурационным файлом **SQLNET.ORA.** Сервер Oracle ищет файл **SQLNET.ORA так** же, как и файл **TNSNAMES.ORA.** Учтите, что он необязательно находится в том же каталоге, что и файл **TNSNAMES.ORA,** — он может быть и в других местах. Если установлен любой из параметров:

- names.directory_path
- names.default_domain

необходимо убедиться, что файл **TNSNAMES.ORA** им соответствует.

Если параметр **names.default_domain** имеет непустое значение, например **(WORLD),** необходимо проверить, указан ли этот домен в соответствующей записи файла **TNSNAMES.ORA.** Вместо **EXTPROC_CONNECTION_DATA,** в качестве имени записи в этом случае должно быть указано **EXTPROC_CONNECTION_DATA.WORLD.**

Если установлен параметр **names.directory_path,** необходимо убедиться, что он содержит значение **TNSNAMES.** Если параметр **names.directory_path** имеет, например, значение **(HOSTNAME,ONAMES),** то протокол Net8 будет использовать имена хостов (Host

Naming Method) при обработке строки подключения **EXTPROC_CONNECTION_DATA**, а если найти параметры подключения таким образом не удастся, — обратится к серверу Oracle Names Server. Поскольку ни один из этих методов не позволит найти запись **EXTPROC_CONNECTION_DATA**, подключение не будет установлено, и вызов extproc завершится неудачно. Просто добавьте в этот список слово **TNSNAMES**, чтобы сервер Oracle мог найти запись **EXTPROC_CONNECTION_DATA** в локальном файле **TNSNAMES.ORA**.

Проверка процесса прослушивания

Проблемы с прослушиванием похожи на проблемы с установкой среды для сервера. При проверке конфигурации процесса прослушивания обратите внимание на следующее:

- используется ли соответствующий конфигурационный файл **LISTENER.ORA**?
- правильно ли настроен этот файл?

Снова, как и при проверке среды сервера, необходимо убедиться, что для прослушивания используется соответствующая среда, позволяющая ему найти требуемый файл **LISTENER.ORA**. Возникают те же проблемы, связанные с поиском конфигурационных файлов в разных местах. В случае сомнений, какой именно набор конфигурационных файлов используется, следует установить соответствующее значение переменной среды **TNS_ADMIN** перед запуском процесса прослушивания. Это гарантирует использование соответствующих конфигурационных файлов. Убедившись, что используются нужные конфигурационные файлы, необходимо проверить содержимое файла **LISTENER.ORA** в соответствии с представленной ранее информацией.

После этого проверка из среды сервера доступности службы **extproc_connection_data** (с добавлением стандартного домена) с помощью программы **tnsping** должна завершиться успешно. Например, у меня используется стандартный домен **us.oracle.com**, и проверка доступности дает следующий результат:

```
C:\oracle\network\ADMIN>tnsping extproc_connection_data.us.oracle.can

TNS Ping Utility for 32-bit Windows: Version 8.1.6.0.0 - Production on
06-AUG-2000 09:34:32

(c) Copyright 1997 Oracle Corporation. All rights reserved.

Attempting to contact (ADDRESS=(PROTOCOL=IPC)(KEY=EXTPROC1))
OK (40 msec)
```

Это подтверждает, что среда сервера базы данных и процесса прослушивания сконфигурирована правильно. Не должно быть никаких сообщений об ошибке **ORA-28575: unable to open RPC connection to external procedure agent (ORA-28575: невозможно открыть соединение RPC с агентом внешней процедуры).**

Первая проверка

Рекомендуется проверить правильность выполнения внешних процедур с помощью демонстрационной программы. Для этого имеются две причины.

- Служба поддержки Oracle поможет настроить/сконфигурировать демонстрационную программу. Если служба поддержки работает со знакомым примером, то сможет решить любые проблемы намного быстрее.
- Предлагаемая демонстрационная программа демонстрирует правильный подход к компиляции и компоновке на данной платформе.

Демонстрационная программа находится в каталоге **[ORACLE_HOME]/plsql/demo** во всех версиях Oracle 8i. Шаги, которые необходимо выполнить для создания демонстрационной программы, описаны в следующих разделах.

Компиляция кода extproc.c

Сначала необходимо скомпилировать код **extproc.c** в DLL-библиотеку или файл .so/ ,sl/.a. Чтобы сделать это в Windows, достаточно перейти в каталог **ORACLE_HOME\ plsql\demo** и набрать **make** (в этом каталоге корпорация Oracle поставляет файл **make.bat):**

```
C:\oracle\plsql\demo>make
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 10.00.5270 for 80x86
Copyright (C) Microsoft Corp 1984-1995. All rights reserved.
extproc.c
Microsoft (R) 32-Bit Incremental Linker Version 3.00.5270
Copyright (C) Microsoft Corp 1992-1995. All rights reserved.

/out:extproc.dll
/dll
/implib:extproc.lib
/debug
..\..\oci\lib\msvc\oci.lib
msvcrt.lib
/nod:libcmt
/DLL
/EXPORT:UpdateSalary
/EXPORT:PercentComn
/EXPORT:PercentComm_ByRef
/EXPORT:EmpExp
/EXPORT:CheckEmpName
/EXPORT:LobDemo
extproc.obj
C:\oracle\plsql\demo>
```

В ОС UNIX делается практически то же самое, но для компиляции необходимо ввести другую команду. Вот как она должна выглядеть:

```
$ make -f demo_plsql.mk extproc.so
/usr/ccs/bin/make -f /export/home/ora816/rdbms/demo/demo_rdbms.mk
extproo_callback SHARED_LIBNAME=extproc.so OBJS="extproc.o"
```

После завершения работы команды будет получен файл с расширением **.dll** в Windows или с расширением **.so/.sl/.a** в ОС UNIX; расширение зависит от платформы. Например, в ОС Solaris используется расширение **.so,** а **в** ОС **HP/UX — .si.**

Настройка учетной записи SCOTT/TIGER

Чтобы эта демонстрационная программа работала правильно, необходимо создать демонстрационную учетную запись SCOTT/TIGER. Если в базе данных нет учетной записи SCOTT/TIGER, ее можно создать с помощью оператора:

```
SQL> grant connect, resource to scott identified by tiger;
```

При этом создается пользователь SCOTT, которому предоставляются привилегии подключения и создания объектов (таблиц, пакетов и т.п.). Имеет смысл задать для этого пользователя другое стандартное табличное пространство вместо SYSTEM, а также явно задать временное табличное пространство.

```
SQL> alter user scott default tablespace tools temporary tablespace temp;
```

При наличии учетной записи **SCOTT/TIGER**, прежде чем продолжать, необходимо предоставить ей еще одну дополнительную привилегию. Пользователю **SCOTT** необходима привилегия **CREATE LIBRARY**. Она позволит ему выполнить оператор **CREATE LIBRARY**, необходимый для использования внешних процедур. К этому оператору мы еще вернемся. Поскольку привилегия эта — весьма мощная, имеет смысл отобрать ее у пользователя **SCOTT** после выполнения примера. Для предоставления привилегии необходимо выполнить следующий оператор:

```
SQL> grant create library to SCOTT;
```

подключившись как один из пользователей, имеющих привилегию **CREATE LIBRARY** с опцией **ADMIN** (например, как пользователь **SYSTEM** или любой другой пользователь, которому предоставлена роль **DBA**).

Наконец, необходимо убедиться, что в схеме **SCOTT** созданы и наполнены данными демонстрационные таблицы **EMP/DEPT**. Это можно проверить следующим образом:

```
SQL> select count(*) from emp;

  COUNT(*)

        14

SQL> select count(*) from dept;

  COUNT(*)

         4
```

Если эти таблицы не существуют или они не заполнены, их можно пересоздать, выполнив сценарии **demodrop.sql** (для удаления таблиц) и **demobld.sql** (для их создания и наполнения данными). Эти сценарии находятся в каталоге **[ORACLE_HOME]\sqlplus\demo** и должны выполняться в среде SQL*Plus при подключении от имени пользователя **SCOTT.**

Создание библиотеки demolib

Следующий шаг — создание объекта-библиотеки в базе данных Oracle. Этот объект просто сопоставляет имя библиотеки (любое имя длиной до 30 символов) с физическим

файлом операционной системы. Этим файлом операционной системы является созданный нами скомпилированный двоичный файл. Пользователь, выполняющий оператор **CREATE LIBRARY,** должен обладать привилегией **CREATE LIBRARY,** предоставленной ему непосредственно или через роль. Эта привилегия считается достаточно мощной и должна предоставляться только тем, кто пользуется доверием. Эта привилегия позволяет пользователям выполнять C-код на сервере с правами той учетной записи, от имени которой работает служба **extproc.** Это одна из причин, почему необходимо конфигурировать службу **extproc** так, чтобы она работала не от имени пользователя — владельца программного обеспечения Oracle (чтобы избежать случайного или намеренного переписывания, например, табличного пространства SYSTEM).

Для выполнения этого шага необходимо ввести в среде SQL*Plus команды:

```
SQL> connect scott/tiger
Connected.
SQL> create or replace library demolib as
  2  'c:\oracle\plsql\demo\extproc.dll';
  3  /
Library created.
```

Имя **DEMOLIB** выбрано для библиотеки разработчики демонстрационной программы; имя **DEMOLIB** использовать обязательно. Имя файла, **c:\oracle\pisql\demo\extproc.dll**, может отличаться (я создавал пример непосредственно в демонстрационном каталоге в **ORACLE_HOME**). У вас может отличаться и значение **ORACLE_HOME**; демонстрационную программу можно создавать вообще в другом каталоге. В операторе создания библиотеки необходимо указать фактическое местонахождение файла extproc.dll, созданного на первом шаге.

Установка и запуск

Последний шаг демонстрации — установка PL/SQL-кода, создающего соответствующие подпрограммы для функций в библиотеке demolib. Нам интересен не столько их исходный код, сколько результат их выполнения. Цель демонстрации — протестировать работу внешних процедур. Как их создавать и подключать, мы рассмотрим далее.

Теперь просто выполняем команды:

```
SQL> connect scott/tiger
Connected.
SQL> @extproc
```

перейдя предварительно в каталог **[ORACLE_HOME]\plsql\demo.** Вы должны получить примерно следующее:

```
SQL> @extproc

Package created.

No errors.

Package body created.

No errors.
ENAME      : ALLEN
JOB        : SALESMAN
```

```
SALARY     : 1600
COMMISSION : 300
Percent Commission : 18.75
ENAME      : MARTIN
JOB        : SALESMAN
SALARY     : 1250
COMMISSION : 1400
Percent Commission : 112
Return value from CheckEmpName : 0
old_ename value on return      : ANIL
ENAME      : 7369
HIREDATE   : 17-DEC-80
Employee Experience Test Passed.
*************************************

PL/SQL procedure successfully completed.
... (здесь будут и другие информационные сообщения)!
```

Это показывает, что внешние процедуры правильно сконфигурированы и могут использоваться в системе. Первая процедура выполняет многие из функций созданной нами библиотеки **extproc.dll**. Поэтому можно сделать вывод, что все сконфигурировано правильно.

Если система сконфигурирована неправильно, вы скорее всего получите следующее:

```
SQL> @extproc

Package created.

No errors.

Package body created.

No errors.
BEGIN demopack.demo_procedure; END;
*
ERROR at line 1:
ORA-28575: unable to open RPC connection to external procedure agent
ORA-06512: at "SCOTT.DEMOPACK", line 61
ORA-06512: at "SCOTT.DEMOPACK", line 103
ORA-06512: at line 1
```

Это означает, что надо перечитать представленный ранее раздел "Конфигурирование сервера" и выполнить все описанные в нем проверки.

Наша первая внешняя процедура

Предполагая, что среда разработки настроена, как описано выше, и готова к использованию внешних процедур, попробуем создать первую собственную внешнюю процедуру. В этом примере мы просто будем передавать переменные различных типов (строки, числа, даты, массивы и т.д.) и рассмотрим, как будет выглядеть соответствующий код на языке C для получения этих значений. Внешняя процедура будет обрабатывать некоторые из этих значений, изменяя значения параметров, переданных в режиме **OUT** или **IN/OUT**, в зависимости от значений других параметров, переданных в режиме **IN** или **IN/OUT**.

Я продемонстрирую свой способ сопоставления для этих переменных, поскольку есть много различных вариантов сопоставления и полезных приемов. Я покажу метод, который предпочитаю использовать, несмотря на некоторую избыточность, потому что он обеспечивает максимум информации во время выполнения. Кроме того, я представлю шаблон, по которому я создаю хранимые процедуры. Этот шаблон реализует многие конструкции, необходимые в любом реальном приложении:

- **Управление состоянием.** Внешние процедуры могут "потерять" информацию о состоянии (текущие значения статических или глобальных переменных). Это связано с реализованным в **EXTPROC** механизмом кэширования. Поэтому необходимо использовать механизм определения и сохранения состояния в программах на языке C.
- **Механизмы трассировки.** Внешние процедуры выполняются на сервере отдельно от других процессов. Хотя на некоторых платформах эти процедуры можно отлаживать с помощью обычного отладчика, это весьма сложно; если же ошибки возникают только при использовании внешней процедуры одновременно многими пользователями, то просто невозможно. Необходимо средство генерации трассировочных файлов по требованию, "начиная с этого момента". Эти файлы аналогичны трассировочным файлам, которые сервер Oracle генерирует при выполнении оператора **alter session set sql_trace = true;** цель их создания — записать данные времени выполнения в текстовый файл для отладки/настройки.
- **Использование параметров.** Необходимо средство параметризации внешних процедур, чтобы можно было управлять их работой извне, с помощью файла параметров, аналогично тому, как файл **init.ora** используется для управления сервером.
- **Общая обработка ошибок.** Необходимо простое средство выдачи вразумительных сообщений об ошибках пользователю.

Оболочка

Я собираюсь начать с PL/SQL-прототипа. Будут представлены спецификации подпрограмм на PL/SQL, которые планируется реализовать. В этом примере я собираюсь реализовать набор процедур, принимающих параметр в режиме IN и параметр в режиме **OUT** (или **IN/OUT).** Мы напишем такую процедуру для каждого существенного (часто используемого) типа данных. На примере этих процедур я продемонстрирую, как правильно передавать входные данные и получать результаты каждого из этих типов. Кроме того, я хочу создать ряд функций, возвращающих результаты некоторых из этих типов. Мне кажется, наиболее существенны следующие типы:

- строки (размером вплоть до максимально поддерживаемого языком PL/SQL, 32 Кбайта);
- числа (данные типа **NUMBER,** с любым масштабом и точностью);
- даты **(DATE);**
- целые числа (данные типа **BINARY_INTEGER);**

- данные булева типа (BOOLEAN);
- данные типа RAW (размером до 32 Кбайт);
- большие объекты (для всех данных размером >32 Кбайт);
- массивы строк;
- массивы чисел;
- массивы дат.

Для этого необходимо сначала создать несколько типов наборов. Они будут представлять массивы строк, чисел и дат:

```
tkyte@TKYTE816> create or replace type numArray as table of number
  2  /
Type created.

tkyte@TKYTE816> create or replace type dateArray as table of date
  2  /
Type created.

tkyte@TKYTE816> create or replace type strArray as table of varchar2(255)
  2  /
Type created.
```

Теперь можно создавать спецификацию пакета. Она представляет собой набор перегруженных процедур для тестирования передачи параметров. Каждая процедура имеет параметры, передаваемые в режиме **IN** и **OUT**, за исключением версии для данных типа **CLOB**, в которой параметр передается в режиме **IN/OUT**. Клиент должен инициализировать параметр **LOB IN OUT**, а внешняя процедура заполнит этот объект:

```
tkyte@TKYTE816> create or replace package demo_passing_pkg
  2  as
  3      procedure pass(p_in in number, p_out out number);
  4
  5      procedure pass(p_in in date, p_out out date);
  6
  7     procedure pass(p_in in varchar2, p_out out varchar2);
  8
  9     procedure pass(p_in in boolean, p_out out boolean);
 10
 11      procedure pass(p_in in CLOB, p_out in out CLOB);
 12
 13     procedure pass(p_in in numArray, p_out out numArray);
 14
 15     procedure pass(p_in in dateArray, p_out out dateArray);
 16
 17     procedure pass(p_in in strArray, p_out out strArray);
```

Нельзя использовать перегрузку для процедур, использующих параметры типа **RAW** и **INT**, поскольку вызов **PASS(RAW, RAW)** будет совпадать по сигнатуре с **PASS(VARCHAR2,VARCHAR2)**, а **PASS(INT,INT)** - с **PASS(NUMBER,NUMBER)**. Поэтому для этих двух типов данных я в виде исключения создам процедуры со специальными именами:

```
19      procedure pass_raw(p_in in RAW, p_out out RAW);
20
21      procedure pass_int(p_in   in binary_integer,
22                         p_out  out binary_integer);
```

Наконец, реализую несколько функций, возвращающих значения, чтобы продемонстрировать, как они реализуются. Создадим по функции для каждого из основных скалярных типов данных:

```
25      function return_number return number;
26
27      function return_date return date;
28
29      function return_string return varchar2;
30
31  end demo_passing_pkg;
32  /

Package created.
```

Операторы **CREATE TYPE** позволяют создать необходимые типы массивов. С их помощью мы определили новые SQL-типы; **numArray** — вложенная таблица чисел, **dateArray** — вложенная таблица дат и **strArray** — вложенная таблица строк типа **VARCHAR2(255)**. Мы создали также спецификацию пакета, который собираемся реализовать, так что можно приступать к реализации. Я представлю ее шаг за шагом. Начнем с создания библиотеки:

```
tkyte@TKYTE816> create or replace library demoPassing
  2  as
  3    'C:\demo_passing\extproc.dll'
  4  /
Library created.
```

Этот оператор, как и в рассмотренном ранее примере, где проверялась настройка учетной записи **SCOTT/TIGER,** просто определяет для сервера Oracle место хранения библиотеки **demoPassing;** в данном случае она находится в файле **C:\demo_passing\extproc.dll.** Тот факт, что эта DLL-библиотека еще не создана, не имеет значения. Объект-библиотека необходим для компиляции тела PL/SQL-пакета, который мы собираемся создавать. Библиотеку **extproc.dll** мы создадим чуть позже. Оператор создания библиотеки можно успешно выполнить даже при ее отсутствии.

Теперь переходим к телу пакета:

```
tkyte@TKYTE816> create or replace package body demo_passing_pkg
  2  as
  3
  4      procedure pass(p_in    in  number,
  5                     p_out   out number)
  6      as
  7      language C
  8      name "pass_number"
  9      library demoPassing
 10      with context
 11      parameters (
```

```
12                  CONTEXT,
13                  p_in   OCINumber,
14                  p_in   INDICATOR short,
15                  p_out  OCINumber,
16          p_out           INDICATOR Short);
```

Итак, на первый взгляд, все начинается как обычно: оператор **CREATE OR REPLACE PACKAGE** и тело процедуры **PROCEDURE Pass(** ... **) as** ..., но затем появляется отличие от обычной хранимой процедуры. Мы создаем спецификацию вызова, а не PL/SQL-код. Спецификация вызова — это метод сопоставления PL/SQL-типов встроенным типам данных языка, на котором создается внешняя процедура. Например, выше выполнено сопоставление параметра **p_in number** типу данных **OCINumber** языка C. Ниже представлено построчное описание того, что делается в данном случае.

- **Строка 7: language C.** Задаем язык. Создаются внешние процедуры на языке C, хотя можно их создавать и на языке Java (но этому посвящена следующая глава).
- **Строка 8: name "pass_number".** Задаем имя функции на языке C, которую будем вызывать из библиотеки **demoPassing.** Для сохранения регистра символов необходимо использовать идентификатор в кавычках (поскольку для языка C регистр символов имеет значение). Обычно все идентификаторы в Oracle переводятся в верхний регистр, но если взять их в двойные кавычки, регистр символов при записи в базу данных будет сохранен. Это имя должно точно совпадать с именем C-функции, вплоть до регистра символов.
- **Строка 9: library demoPassing.** Указываем имя библиотеки, которая будет содержать соответствующий код. Это имя совпадает с именем, заданным в представленном ранее операторе **CREATE LIBRARY.**
- **Строка 10: with context.** Хотя это и не обязательно, я всегда передаю *контекст*. Этот контекст необходим для выдачи осмысленных сообщений об ошибках и использования функций OCI или Pro*C.
- **Строка 11: parameters.** Начало списка параметров. Мы явно задаем последовательность и типы передаваемых параметров. Это, как и передавать контекст, делать не обязательно, но я всегда стараюсь описывать параметры явно. Вместо использования стандартных соглашений и периодического их "угадывания" я явно сообщаю серверу Oracle, какие значения и в каком порядке ожидаю получить.
- **Строка 12: CONTEXT.** Это ключевое слово, стоящее в списке параметров первым, требует от сервера Oracle передать в качестве первого параметр типа **OCIExtProcContext** *. Это ключевое слово можно задавать в любом месте списка параметров, но обычно его указывают первым. **OCIExtProcContext** — тип данных, определяемый функциональным интерфейсом OCI и представляющий информацию о сеансе на сервере.
- **Строка 13: p_in OCINumber. Я** сообщаю серверу Oracle, что следующий параметр в моей C-функции должен быть типа **OCINumber.** В данном случае он будет передаваться как **OCINumber** *, указатель на данные типа **OCINumber.** Таблицу соответствий типов данных см. далее.

- **Строка 14: p_in INDICATOR short.** Я сообщаю серверу Oracle, что следующий параметр в C-функции — *индикаторная переменная* типа **short,** которая позволит определить, имеет ли параметр **p_in** значение NULL. Хотя это необязательно, я всегда передаю индикаторную переменную вместе с каждым параметром. Если этого не делать, нельзя будет определить во внешней процедуре, передано ли значение NULL, или вернуть из нее значение NULL.
- **Строка 15: p_out OCINumber. Я** сообщаю серверу Oracle, что следующий после индикаторной процедуры параметр функции тоже имеет тип **OCINumber. В** данном случае он будет передаваться как **OCINumber** *. Таблицу соответствий типов данных см. далее.
- **Строка 16: p_out INDICATOR short. Я** сообщаю серверу Oracle, что следующий параметр — индикаторная переменная для параметра **p_out,** и она должна быть типа **short.** Поскольку параметр **p_out** передается в режиме OUT, эта индикаторная переменная позволит мне сообщить вызывающему, имеет ли параметр, переданный в режиме **OUT,** значение **NULL.** Поэтому этот параметр передается как данные типа **short** * (указатель), чтобы можно было не только читать значение, но и устанавливать его.

С учетом этого, давайте рассмотрим прототип C-функции, соответствующий только что созданной PL/SQL-процедуре. Этот прототип должен иметь следующий вид:

```
18      --void pass_number
19      -- (
20       -- OCIExtProcContext  * ,  /*  1 : контекст  */
21       --        OCINumber  * ,  /*  2 : P_IN  */
22       --            short    ,  /*  3 : P_IN (индикатор)  */
23       --        OCINumber  * ,  /*  4 : P_OUT  */
24       --            short  *    /*  5 : P_OUT (индикатор)  */
25       -- );
```

Закончим обзор создаваемого тела PL/SQL-пакета, сопоставляющего остальные процедуры/функции. Вот процедура, передающая и принимающая даты: они будут сопоставлены типу данных **OCIDate** языка C, предоставляемому функциональным интерфейсом OCI:

```
27     procedure pass(p_in in date, p_out out date)
28     as
29     language C name "pass_date" library demoPassing
30     with context parameters
31      (CONTEXT,
32        p_in   OCIDate, pin   INDICATOR short,
33       p_out  OCIDate, p_out  INDICATOR short);
34
35       -- void pass_date
36       --   (
37       --  OCIExtProcContext  *,  /*  1 : контекст  */
38       --            OCIDate  *,  /*  2 : P_IN  */
39       --              short   ,  /*  3 : P_IN (индикатор)  */
40       --            OCIDate  *,  /*  4 : P_OUT  */
```

```
41       --              short  *   /*   5 : P_OUT (индикатор) */
42       --    );
```

Давайте рассмотрим, как передавать и принимать данные типа **varchar2** — в этом случае сервер будет сопоставлять строки типу данных **char** * — указателю на строку символов.

```
45     procedure pass(p_in in varchar2, p_out  out  varchar2)
46     as
47     language C name "pass_str" library demoPassing
48    with context parameters
49     (CONTEXT,
50      p_in   STRING, p_in INDICATOR short,
51     pout  STRING, p_out INDICATOR short, p_out MAXLEN int);
52
53      -- void pass_str
54      --    (
55     --  OCIExtProcContext  *,  /*  1 : контекст  */
56      --               char  *,  /*  2 : P_IN  */
57      --            short   ,  /*  3 : P_1N (индикатор)  */
58      --              char  *,  /*  4 : P_OUT  */
59      --            short  *,  /*  5 : P_OUT (индикатор)  */
60      --              int  *   /*  6 : P_OUT (максимальная длина) */
61       --    );
```

В этом коде мы впервые столкнулись с использованием параметра MAXLEN. Он требует от сервера Oracle передать внешней процедуре максимальный размер передаваемого в режиме **OUT** параметра **p_out.** Поскольку мы возвращаем строку, важно знать ее максимально возможную длину, чтобы предотвратить перезапись буфера. Для всех строковых типов, передаваемых в режиме **OUT,** я настоятельно рекомендую использовать параметр **MAXLEN**.

Теперь давайте рассмотрим, как передавать тип PL/SQL **BOOLEAN,** которому будет сопоставляться тип **int** языка C:

```
64    procedure pass(p_in in boolean, p_out out boolean)
65     as
66     language C name "pass_bool" library demoPassing
67    with context parameters
68     (CONTEXT,
69      p_in  int, p_in  INDICATOR short,
70      p_out int, p_out INDICATOR short);
71
72      -- void pass_bool
73      --    (
74     --  OCIExtProcContext  *,  /*   1 : контекст  */
75      --              int   ,  /*   2 : P_IN  */
76      --            short   ,  /*   3 : P_IN (индикатор)  */
77      --              int  *,  /*   4 : P_OUT  */
78      --            short  *   /*   5 : P_OUT (индикатор)  */
79       --    );
```

Рассмотрим пример для типа данных **CLOB.** Мы передаем тип данных PL/SQL **CLOB** как тип данных **OCILobLocator** языка C. Обратите внимание, что в этом случае если

параметр передается в режиме **OUT,** функция должна принимать указатель на указатель. Это позволяет в функции на языке C изменять не только содержимое, на которое указывает локатор большого объекта, но и сам этот локатор, т.е. при необходимости сослаться на другой большой объект:

```
83      procedure pass(p_in in clob, p_out in out clob)
84      as
85      language C name "pass_clob" library demoPassing
86      with context parameters
87      (CONTEXT,
88        p_in   OCILobLocator, p_in   INDICATOR short,
89        p_out  OCILobLocator, p_out  INDICATOR short);
90
91        -- void pass_clob
92        --    (
93        --  OCIExtProcContext  *,  /*   1 : контекст  */
94        --      OCILobLocator  *,  /*   2 : P_IN  */
95        --              short   ,  /*   3 : P_IN (индикатор)  */
96        --      OCILobLocator **,  /*   4 :  P_OUT  */
97        --              short  *   /*   5 : P_OUT (индикатор)  */
98        --    );
```

Затем следуют три процедуры, передающие и принимающие массивы, данные типа наборов в Oracle. Поскольку сопоставления типов во всех трех случаях очень похожи, рассмотрим все их одновременно. C-функции для этих трех процедур имеют абсолютно одинаковые прототипы — каждая получает данные типа **OCIColl,** независимо от передаваемого типа набора:

```
100     procedure pass(p_in in numArray, p_out out numArray)
101     as
102     language C name "pass_numArray" library demoPassing
103     with context parameters
104     (CONTEXT,
105       p_in OCIColl, p_in  INDICATOR short,
106      p_out OCIColl, p_out INDICATOR short);
107
108       -- void pass_numArray
109       --    (
110       --  OCIExtProcContext  *,   /*  1 : контекст  */
111       --            OCIColl  *,   /*   2 : P_IN  */
112       --             short   ,  /*   3 : P_IN (индикатор)  */
113       --            OCIColl **,   /*   4 :  P_OUT  */
114       --              short  *   /*   5 : P_OUT (индикатор)  */
115       --    );
116
117     procedure pass(p_in in dateArray, p_out out dateArray)
118     as
119     language C name "pass_dateArray" library demoPassing
120     with context parameters
121     (CONTEXT,
122       p_in  OCIColl, p_in  INDICATOR short,
123      p_out OCIColl, p_out INDICATOR short);
```

```
124
125     procedure pass(p_in in strArray, p_out out strArray)
126      as
127     language C name "pass_strArray" library demoPassing
128     with context parameters
129      (CONTEXT,
130       p_in  OCIColl, p_in  INDICATOR short,
131       p_out OCIColl, p_out INDICATOR short);
```

Далее следует процедура, передающая и принимающая данные типа **RAW**. В данном случае используется как атрибут **MAXLEN** (который уже использовался в представленной выше процедуре с параметрами типа **VARCHAR2**), так и атрибут **LENGTH**. Необходимо передавать длину данных типа **RAW**, поскольку они содержат двоичную информацию, включая нули, поэтому фактическую длину строки в программе на языке C определить невозможно, и сервер Oracle не сможет понять, какого объема данные возвращаются. Для данных типа **RAW** оба атрибута, **LENGTH** и **MAXLEN**, принципиально важны. Атрибут **LENGTH** передавать обязательно, а **MAXLEN** — желательно.

```
134      procedure pass_raw(p_in in raw, p_out out raw)
135      as
136     language C name "pass_raw " library demoPassing
137     with context parameters
138      (CONTEXT,
139      p_in  RAW, p_in  INDICATOR short, p_in  LENGTH  int,
140      p_out RAW, p_out INDICATOR short, p_out MAXLEN int,
141      p_out LENGTH int);
142      -- void pass_long_raw
143      --   (
144      --  OCIExtProcContext  *,  /*  1 : контекст  */
145      --      unsigned char  *,  /*  2 : P_IN  */
146       --             short   ,  /*  3 : P_IN (индикатор)  */
147       --               int   ,  /*  4 : P_IN (длина)  */
148      --      unsigned char  *,  /*  5 : P_OUT  */
149       --             short  *,  /*  6 : P_OUT (индикатор)  */
150       --               int  *,  /*  7 : P_OUT (максимальная длина) */
151       --                int  *   /*  8 : P_OUT (длина)  */
152       --   );
```

Далее идет процедура, принимающая и передающая в функцию языка C данные PL/SQL типа **BINARY_INTEGER**. В этом случае тип **BINARY_INTEGER** сопоставляется встроенному типу данных int языка C:

```
154    procedure pass_int(p_in in binary_integer, p_out out binary_integer)
155    as
156     language C name "pass_int" library demoPassing
157     with context parameters
158      (CONTEXT,
159       p_in  int, p_in  INDICATOR short,
160       p_out int, p_out INDICATOR short);
161
162        -- void pass_int
163        --   (
```

```
164    --    OCIExtProcContext  *,  /*   1 : контекст */
165            --           int   ,  /*   2 : P_IN */
166           --          short   ,  /*   3 : P_IN (индикатор) */
167            --           int  *,  /*   4 : P_OUT */
168       --              short  *   /*   5 : P_OUT (индикатор) */
169     --     );
```

Ниже представлены оболочки для трех функций, возвращающих числа, даты и строки. В них используется новое ключевое слово **RETURN.** При сопоставлении с функцией необходимо использовать ключевое слово **RETURN** в качестве последнего параметра в списке параметров. При этом задается тип возвращаемого значения, а не формального параметра функции. Ниже я буду указывать три параметра в SQL-оболочке и только два из них в используемом прототипе функции на языке C. Параметр **RETURN OCINumber** задает тип данных, которые **будут возвращаться** в соответствии с прототипом функции **OCINumber *return_number. Я** включил индикаторную переменную даже для возвращаемого значения, чтобы можно было при необходимости вернуть значение **NULL.** Если не включить в список параметров индикаторную переменную, вернуть значение **NULL** будет невозможно. Как было показано в примере для строк, можно также возвращать атрибут **LENGTH**, но не **MAXLEN**, поскольку этот атрибут можно задавать только для параметров, передаваемых в режиме **OUT,** память для которых выделяет сервер Oracle. При возврате значений, поскольку за выделение памяти отвечает разработчик, атрибут **MAXLEN** не имеет смысла.

```
173      function return_number return number
174     as
175     language C name "return_number" library demoPassing
176     with context parameters
177     (CONTEXT, RETURN INDICATOR short, RETURN OCINumber);
178
179    --    OCINumber *return_number
180     --     (
181    --     OCIExtProcContext  *,  /*   1 : контекст  */
182           --          short  *   /*   2 : RETURN (индикатор)  */
183     --     );
184
185    function return_date return date
186    as
187    language C name "return_date" library demoPassing
188    with context parameters
189    (CONTEXT, RETURN INDICATOR short, RETURN OCIDate);
190
191    --    OCIDate *return_date
192     --     (
193    --     OCIExtProcContext  *,  /*   1 : контекст  */
194           --          short  *   /*   2 : RETURN (индикатор)  */
195     --     );
196
197    function return_string return varchar2
198    as
199    language C name "return_string" library demoPassing
```

```
200    with context parameters
201    (CONTEXT, RETURN INDICATOR short, RETURN LENGTH int, RETURN STRING);
202
203         -- char *return_string
204         --    (
205         --  OCIExtProcContext  *,   /*   1 : контекст  */
206         --              short  *,   /*   2 : RETURN (индикатор)  */
207         --                int  *    /*   3 : RETURN (длина)  */
208         --    ) ;
209
210  end demo_passing_pkg;
211  /

Package body created.
```

В последней функции я использую как атрибут **LENGTH,** так и индикаторную переменную. Так я могу передать серверу Oracle длину возвращаемой строки.

Итак, мы имеем:

- спецификацию пакета, определяющую его основные функции;
- набор **новых** типов массивов **SQL;**
- объект-библиотеку в базе данных, задающую соответствие для еще не созданной библиотеки **extproc.dll;**
- тело пакета, представляющее собой спецификации SQL для функций на языке C. Эти спецификации сообщают серверу Oracle, какие данные (индикаторные переменные, параметры, контексты и т.д.) и как (на уровне типа данных) передавать внешней процедуре.

Теперь, после рассмотрения примера, имеет смысл представить таблицы соответствия типов данных. Одна таблица показывает, какие внешние типы данных можно использовать для SQL-типа X. Затем для выбранного внешнего типа, показано, какой тип данных языка C будет фактически использоваться. Таблицы соответствия типов данных можно найти в руководстве *Oracle Application Developers Guide — Fundamentals;* здесь они представлены просто для справки. Учтите, что внешний тип данных — это не тип, используемый в языке C (и не тип, используемый в языке SQL или PL/SQL). Для определения фактического типа данных, который должен использоваться в языке C, обратитесь ко второй таблице.

Тип данных SQL или PL/SQL	*Внешний тип данных*	*Стандартный тип*
BINARY INTEGER, BOOLEAN, PLS_INTEGER	[unsigned]char, [unsigned]short, [unsigned]int, [unsigned]long, sb1, sb2, sb4, ub1, ub2, ub4, size_t	int
NATURAL, NATURALN, POSITIVE, POSITIVEN, SIGNTYPE	[unsigned]char, [unsigned]short, [unsigned]int, [unsigned]long, sb1, sb2, sb4, ub1, ub2, ub4, size_t	unsigned int
FLOAT, REAL	Float	float
DOUBLE PRECISION	Double	double

Тип данных SQL или PL/SQL	*Внешний тип данных*	*Стандартный тип*
CHAR, CHARACTER, LONG, NCHAR, NVARCHAR2, ROWID, VARCHAR2, VARCHAR	string, ocistring	string
LONG RAW, RAW	raw, ociraw	raw
BFILE, BLOB, CLOB, NCLOB	ociloblocator	ociloblocator
NUMBER, DEC, DECIMAL, INT, INTEGER, NUMERIC, SMALLINT	[unsigned]char, [unsigned]short, [unsigned]int, [unsigned]long, sb1. sb2, sb4, ub1, ub2, ub4. size_t, ocinumber	ocinumber
DATE	Ocidate	ocidate
Абстрактные типы данных (АТД)	Dvoid	dvoid
Наборы (вложенные таблицы, массивы VARRAY)	Ocicoll	ocicoll

Итак, в таблице приводятся внешние типы данных, соответствующие типу данных SQL или PL/SQL. Я рекомендую использовать стандартные типы, поскольку с ними проще всего работать в функциях на языке C. Внешний тип данных очень похож на тип данных языка C, но надо пойти на шаг дальше. Поскольку любой тип данных SQL или PL/SQL может задаваться для параметра, передаваемого в режиме in, in out, out или определять тип возвращаемого значения функции, при определении фактического типа данных языка C необходимо дополнительное уточнение. В общем случае параметры, которые возвращаются или передаются в режиме in, передаются по значению, а параметры в режимах in out, out или передаваемые по ссылке явно, передаются через указатели, по ссылке. В следующей таблице показано, какой тип данных в C надо использовать для соответствующего внешнего типа данных и режима передачи параметра:

Внешний тип данных	*Тип в языке C для параметров IN и возвращаемых значений*	***Тип в** языке C для параметров IN OUT, OUT **и** при передаче по ссылке*
[unsigned] char	[unsigned] char	[unsigned] char *
[unsigned] short	[unsigned] short	[unsigned] short *
[unsigned] int	[unsigned] int	[unsigned] int •
[unsigned] long	[unsigned] long	[unsigned] long *
size t	size t	size t *
sb1	sb1	sb1 *
sb2	sb2	sb2 *
sb4	sb4	sb4 *
ub1	ub1	ub1 *
ub2	ub2	ub2 *
ub4	ub4	ub4 *

Внешний тип данных	***Тип в*** *языке C для параметров IN и возвращаемых значений*	***Тип*** *в языке C для параметров IN OUT, OUT и при передаче по ссылке*
float	float	float *
double	double	double *
string	char *	char *
raw	unsigned char *	unsigned char *
Ociloblocator	OCILobLocator *	OCILobLocator **
Ocinumber	OCINumber *	OCINumber *
Ocistring	OCIString *	OCIString *
Ociraw	OCIRaw •	OCIRaw *
Ocidate	OCIDate •	OCIDate *
Ocicoll	OCIColl *	OCIColl **
АТД	dvoid *	dvoid •

Код на языке C

Теперь можно переходить к реализации библиотеки на языке C. Начнем с общего шаблона, который я использую для внешних процедур. Этот шаблон включает стандартные заголовочные файлы, заголовочный файл Oracle OCI и три функции: **debugf, oci_error** и **raise_application_error.** Эти функции обеспечивают поддержку механизма трассировки **(debugf)** и общей обработки ошибок **(oci_error** и **raise_application_error).** Я просто копирую этот файл в каждый новый проект создания внешних процедур и начинаю разработку с него.

```
#include<stdio.h>
#include <stdlib.h>
#include<time.h>
#include <string.h>
#include <errno.h>
#include<ctype.h>


#include<oci.h>

#ifdef WIN_NT
#define INI_FILE_NAME "c:\\WtempWextproc.ini"
#else
#define INI_FILE_NAME "/tmp/extproc.ini"
#endif

#definestrupr(a) {char * cp; for(cp=a;*cp;*cp=toupper(*cp), cp++);}
```

Выше представлено начало используемого мною шаблона на языке C. Я включаю заголовочные файлы, а также задаю местонахождение файла параметров. Есть много способов задать местонахождение этого файла на этапе выполнения. Например, если бы внешняя процедура создавалась для выполнения в среде Windows, можно было бы ис-

пользовать функции Windows **RegOpenKeyEx, RegQueryInfoKey** и **RegEnumValue** для получения сведений о местонахождении файла параметров из системного реестра. В среде UNIX можно использовать переменную среды. В этом примере я просто явно указываю местонахождение в коде программы. Это вполне допустимо, поскольку можно потребовать, чтобы параметры инициализации помещались в известный стандартный файл (например, в файл /etc/имя_внешней_процедуры.ora в ОС UNIX или **c:\имя_внешней_процедуры\имя_внешней_процедуры.ora** в Windows).

Теперь переходим к самому коду. В следующей части определяется контекст. Он содержит то, что в обычной программе принято задавать в глобальных переменных. Мы не можем использовать глобальные переменные во внешней процедуре, поскольку это абсолютно ненадежно. Кроме того, поскольку статические данные будут инициализироваться при каждом вызове, использование глобальных переменных в любом случае будет некорректным. Для получения и установки контекста внешней процедуры мы будем использовать средства функционального интерфейса управления контекстом библиотеки **OCI.** В представленную ниже структуру можно добавлять любые переменные, содержащие информацию о состоянии, которую нужно сохранять между вызовами.

Я определил следующие глобальные переменные.

- **OCIExtProcContext * ctx.** Контекст, который передается каждой внешней процедуре. Он потребуется во многих случаях, в частности при обработке ошибок.
- **OCIEnv * envhp.** Указатель на среду **OCI.** Он понадобится практически при каждом обращении к функции OCI.
- **OCISvcCtx * svchp.** Дескриптор службы **OCI.** Он понадобится во многих (но не во всех) вызовах функций **OCI.**
- **OCIError * errhp.** Дескриптор ошибки OCI. Он будет использоваться практически во всех вызовах функций **OCI** для обработки возможных ошибок.
- **int curr_lineno** и **char * curr_filename.** Эти переменные будут использоваться процедурой трассировки. Имя файла исходного кода и номер строки, из которой вызвана процедура трассировки, будут запоминаться, чтобы при выдаче сообщения было понятно, из какой строки и какого файла было выдано сообщение. Это пригодится при отладке "с нуля". В главе 10 я писал о снабжении кода средствами трассировки и отладки — во внешних процедурах это особенно важно.
- **ub1 debugf_flag.** Флаг, определяющий, выдаются ли трассировочные сообщения. Если этот флаг не установлен, мы "проигнорируем" обращения к **debugf** (функция **debugf** представлена ниже). Это позволит оставить вызовы функции трассировки в коде и при передаче его в производственную эксплуатацию, чтобы при необходимости трассировку легко можно было включить.
- **char debugf_path[255].** Эта переменная задает каталог, в который будут выдаваться отладочные сообщения.
- **char debugf_filename[50].** Эта переменная задает имя файла отладочных сообщений в указанном выше каталоге.

```
typedef struct myCtx
{
  OCIExtProcContext * ctx;     /* Контекст, передаваемый внешний
                                  /* процедурам */
  OCIEnv *            envhp;   /* Дескриптор среды OCI */
  OCISvcCtx *         svchp;   /* Дескриптор службы OCI */
  OCIError *          errhp;   /* Дескриптор ошибки OCI */

  int                curr_lineno;
  char *             curr_filename;

  ub1                debugf_flag;
  char               debugf_path[255];
  char               debugf_filename[50];

  /* добавьте сюда необходимые переменные состояния..._*/
}
   myCtxStruct;
```

Затем в шаблоне исходного кода следует функция **debugf** — процедура трассировки. Это С-функция, работающая аналогично стандартной функции **fprintf** и даже принимающая любое количество аргументов (обратите внимание на троеточие в списке аргументов). Первый ее аргумент — контекст — описанная выше структура, предоставляющая информацию о состоянии. Я всегда предполагаю, что указатель на состояние имеет имя **myCtx** (в макросе для функции **debugf** это предположение используется). Функция **debugf** демонстрирует кое-что новое. В ней представлена большая часть функций библиотеки OCI для работы с файлами, которые очень похожи на семейство функций **fopen/fread/fwrite/fclose** языка С. Код функции **debugf,** который вызывается, только если установлен флаг **myCtx->debugf_flag,** открывает файл, формирует сообщение, записывает его и закрывает файл.

Этот код может служить примером того, как используется контекст. Контекст содержит информацию о состоянии сеанса и важные переменные, например структуры **OCIEnv** и **OCIError,** необходимые для всех вызовов функций OCI. Показано, как изменять состояние, манипулируя переменными в структуре контекста (как это делает макрос **debugf).** Макрос **debugf** позволяет "проигнорировать" обращения к реализующей трассировку функции **_debugf().** Дело в том, что если либо контекст **myCtx,** либо флаг **myCtx->debugf_flag** не установлены, состояние контекста никогда не изменяется, и функция **_debugf()** никогда не вызывается. Это означает, что можно оставить отладочные операторы в производственном коде, поскольку их наличие не повлияет на производительность в долгосрочной перспективе (пока флаг **debugf_flag** имеет значение **false).**

```
void _debugf(myCtxStruct * myCtx, char * fmt, ...)
{
va_list         ap;
OCIFileObject * fp;
time_t          theTime = time(NULL);
char            msg[8192];
ub4             bytes;
    if (OCIFileOpen(myCtx->envhp, myCtx->errhp, &fp,
                    myCtx->debugf_filenaroe,
                    myCtx->debugf_path,
```

```
                    OCI_FILE_WRITE_ONLY, OCI_FILE_APPEND|OCI_FILE_CREATE,
                    OCI_FILE_TEXT) != OCI_SUCCESS) return;

    strftime(msg, sizeof(msg),
             "%y%m%d %H%M%S GMT ", gmtime(&theTime));
    OCIFileWrite(myCtx->envhp, myCtx->errhp, fp, msg, strlen(msg), &bytes);

    va_start(ap,fmt);
    vsprintf(msg, fmt, ap);
    va_end(ap);
    strcat(msg,"\n");

    CCIFileWrite(myCtx->envhp, myCtx->errhp, fp, msg, strlen(msg), (bytes);
    OCIFileClose(myCtx->envhp, myCtx->errhp, fp);
)
```

Следующий фрагмент кода представляет интерфейсный макрос для функции debugf. Этот макрос использовать удобнее, чем функцию _debugf. Вместо обязательной передачи значений **_LINE** , **_FILE** при каждом вызове, достаточно написать:

```
debugf( myCtx, "This is some format %s", some_string );
```

и этот макрос автоматически установит соответствующие значения в контексте, а затем вызовет функцию **_debugf**.

```
void _debugf(myCtxStruct * myCtx, char * fmt, . . . ) ;
#define debugf  \
if ((myCtx!=NULL) && (myCtx->debugf_flag)) \
    myCtx->curr_lineno =__LINE__,  \
    myCtx->curr_filename = __FILE__, \
    _debugf
```

После этого в шаблоне следует утилита **raise_application_error** для обработки ошибок. Это имя, конечно, знакомо каждому разработчику, использовавшему язык PL/SQL. **raise_application_error** — встроенная функция PL/SQL, возбуждающая исключительную ситуацию в случае ошибки. Эта функция у нас имеет такое же назначение. Если внешняя процедура вызывает эту функцию раньше, чем завершает работу, возвращаемые внешней процедурой значения игнорируются, а вместо этого в вызывающей среде возбуждается исключительная ситуация. Это позволяет обрабатывать ошибки, возникающие во внешней процедуре, точно так же, как и в любой PL/SQL-процедуре.

```
static int raise_application_error(myCtxStruct * myCtx,
                                   int           errCode,
                                   char *        errMsg,  ...)
{
char   msg[8192];
va_list ap;

    va_start(ap,errMsg);
    vsprintf(msg, errMsg, ap);
    va_end(ap);

    debugf(myCtx,  "raise application error( %d, %s )", errCode, msg);
    if  (OCIExtProcRaiseExcpWithMsg(myCtx->ctx,errCode,msg,0) ==
                                                  OCIEXTPROC_ERROR)
```

```
    {
      debugf(myCtx, "Не удалось возбудить исключительную ситуацию");
    }
    return -1;
}
```

Дальше следует еще одна функция обработки ошибок, **lastOciError.** Эта функция получает контекст текущего сеанса и, используя его структуру OCIError, извлекает теки сообщения о последней произошедшей ошибке OCI. Этот текст выбирается в память, выделенную с помощью вызова OCIExtProcAllocCallMemory(). Любая область памяти, выделенная этой функцией, будет автоматически освобождена при выходе из внешней процедуры. Эта функция чаще всего используется в обращении к функции **raise_application_error** после неудачного вызова одной из функций OCI. Благодаря ей можно узнать причину возникновения ошибки OCI.

```
static char * lastOciError(myCtxStruct * myCtx)
{
sb4         errcode;
char        * errbuf = (char*)OCIExtProcAllocCallMemory(myCtx->ctx, 256);

    strcpy(errbuf, "unable to retrieve message\n");
    OCIErrorGet(myCtx->errhp, 1, NULL, &errcode, errbuf,
                255, OCI_HTYPE_ERROR);
    errbuf[strlen(errbuf)-1] = 0;
    return errbuf;
}
```

Теперь переходим к "основной" функции в шаблоне для создания внешних процедур; речь идет о функции **init.** Она отвечает за формирование и получение информации о состоянии и обработку параметров, установленных в файле инициализации. Это слишком большая функция, чтобы описывать ее полностью, но достаточно простая, если разобраться с используемыми вызовами функций библиотеки OCI.

Функция init создает структуру myCtxStruct и вызывает необходимые функции инициализации OCI. Вначале функция получает дескрипторы среды OCI. Она делает это одним из двух способов. Если используются только средства OCI (без прекомпилятора Pro*C), достаточно просто вызвать OCIExtProcGetEnv с передачей контекста внешней процедуры. Эта функция OCI автоматически получает все необходимые дескрипторы. Если используется и библиотека OCI, и прекомпилятор Pro*C, применяется конструкция EXEC SQL REGISTER CONNECT :ctx. Она выполняет настройку на уровне Pro*C. При этом все равно необходимо получить дескрипторы среды OCI, но для этого придется использовать вызовы библиотеки Pro*C: SQLEnvGet, SQLSvcCtxGet. Уберите комментарий с используемого метода и закомментируйте другой метод.

```
/*————————это надо включать только для внешних процедур,
использующих средства Pro*C!! ————————
#define SQLCA_INIT
EXEC SQL INCLUDE sqlca;

static myCtxStruct * init(OCIExtProcContext * ctx)
{
ub1         false = 0;
```

```
myCtxStruct *myCtx = NULL;
OCIEnv      *envhp;
OCISvcCtx   *svchp;
OCIError    *errhp;
ub4          key = 1;

    if (OCIExtProcGetEnv( ctx, &envhp, &svchp, &errhp ) != OCI_SUCCESS)
    (
         OCIExtProcRaiseExcpWithMsg(ctx,20000,
                                    "не удалось подключиться к OCI",0);
         return NULL;
    }
/* — замените представленный выше вызов OCIExtProcGetEnv() следующим——
  — при использовании Pro*C ————————————————————————————————————————

    EXEC SQL REGISTER CONNECT USING :ctx;
    if (sqlca.sqlcode < 0)
    {
        OCIExtProcRaiseExcpWithMsg(ctx,20000,sqlca.sqlerrm.sqlernnc,70);
        return NULL;
    }
    if ((SQLEnvGet(0, &envhp) != OCI_SUCCESS) ||
         (OCIHandleAlloc(envhp, (dvoid**)&errhp,
                         OCI_HTYPE_ERROR,0,0) != OCI_SUCCESS) ||
         (SQLSvcCtxGet(0, NULL, 0, &svchp) != OCI_SUCCESS))
    {
         OCIExtProcRaiseExcpWithMsg(ctx,20000,"не удалось получить среду
->OCI",0);
         return NULL;
    }
  ---------- ------------------- -———————————————— --
```

Получив среду OCI, необходимо вызвать функцию OCIContextGetValue() для получения контекста. Эта функция принимает среду OCI и ключ и пытается получить указатель. Ключ в данном случае — 64-битовое число. Можно сохранять сколько угодно контекстов, но в этот раз мы будем использовать один.

```
    if (OCIContextGetValue(envhp, errhp, (ubl*)&key, sizeof(key),
                           (dvoid**)&myCtx) != OCI_SUCCESS)
    {
        OCIExtProcRaiseExcpWithMsg(ctx,20000, "не удалось получить
                                   контекст OCI",0);
        return NULL;
    }
```

При получении указателя **NULL** (это происходит, если контекст еще не установлен), мы выделяем достаточный объем памяти для контекста и сохраняем его в контексте. Для выделения блока памяти, который остается действительным на все время существования процесса, вызывается функция **OCIMemoryAllocate**. После того как память выделена, она сохраняется в контексте с помощью вызова функции **OCIContextSetValue**. Эта функция на время сеанса сопоставляет указатель (который не будет изменяться) выбранному ключу. Следующий вызов функции **OCIContextGetValue** с тем же ключом в пределах того же сеанса позволит получить этот указатель.

```
if (myCtx = NULL)
{
    if (OCIMemoryAlloc(envhp, errhp, (dvoid**)&myCtx,
                       OCI_DURATION_PROCESS,
                       sizeof(myCtxStruct),
                       OCI_MEMORY_CLEARED)!=OCI_SUCCESS)
    {
        OCIExtProcRaiseExcpWithMsg(ctx,20000,
                                   "не удалось получить память OCI",0);
        return NULL;
    }
    myCtx->ctx   = ctx;
    myCtx->envhp = envhp;
    myCtx->svchp = svchp;
    myCtx->errhp = errhp;
    if (OCIContextSetValue(envhp, errhp,
                           OCI_DURATION_SESSION, (ubl*)&key,
                           sizeof(key), myCtx ) != OCI_SUCCESS)
    {
        raise_application_error(myCtx, 20000, "%s",
                                lastOciError(myCtx));
        return NULL;
    )
```

Продолжаем, поскольку получение указателя NULL означает, что параметры еще не обработаны. Мы обработаем их в следующем блоке кода. Для обработки файлов, правила построения которых аналогичны файлу параметров инициализации Oracle, используются предоставляемые библиотекой OCI функции управления параметрами. Файл параметров инициализации описан в главе 2. Я обычно использую этот файл для управления выдачей трассировочной и отладочной информации и задания стандартных значений для других переменных, используемых в программе. Файл параметров инициализации для данного примера может выглядеть следующим образом:

```
debugf = true
debugf_filename=extproc2.log
debugf path = /tmp/
```

Он включает выдачу трассировочной информации **(debugf = true)** в файл **/tmp/extproc2.log.** Можно добавить в этот файл дополнительные параметры и изменить код функции **init** так, чтобы можно было читать их и устанавливать соответствующие значения в контексте сеанса. Процесс чтения и обработки файла параметров инициализации состоит из следующих шагов.

1. Вызов функции OCIExtractInit для инициализации библиотеки обработки параметров.
2. Вызов функции OCIExtractSetNumKeys для передачи библиотеке OCI количества запрашиваемых имен. Оно должно совпадать с количеством параметров в файле параметров.
3. Вызов функции OCIExtractSetKey столько раз, сколько было указано в вызове функции **OCIExtractSetNumKeys().**

4. Вызов функции OCIExtractFromFile для обработки файла параметров.
5. Вызов функции OCIExtractTo<тип данных> для поочередного получения значений параметров.
6. Вызов функции OCIExtractTerm для завершения работы библиотеки обработки параметров и освобождения выделенных ей системных ресурсов.

```
    if ((OCIExtractInit(envhp, errhp) != OCI_SUCCESS) ||
        (OCIExtractSetNumKeys(envhp, errhp, 3) != OCI_SUCCESS) ||
        (OCIExtractSetKey(envhp, errhp, "debugf",
                          OCI_EXTRACT_TYPE_BOOLEAN,
                          0, &false, NULL, NULL) != OCI_SUCCESS) ||
        (OCIExtractSetKey( envhp, errhp, "debugf_filename",
                           OCI_EXTRACT_TYPE_STRING,
                           0, "extproc.log",
                           NULL, NULL) != OCI_SUCCESS) ||
        (OCIExtractSetKey( envhp, errhp, "debugf_path",
                           OCI_EXTRACT_TYPE_STRING,
                           0, "", NULL, NULL) != OCI_SUCCESS) ||
        (OCIExtractFromFile(envhp, errhp, 0,
                            INI_FILE_NAME) != OCI_SUCCESS) ||
        (OCIExtractToBool(envhp, errhp, "debugf", 0,
                          &myCtx->debugf_flag) != OCI_SUCCESS) ||
        (OCIExtractToStr(envhp, errhp, "debugf_filename", 0,
                         myCtx->debugf_filename,
                         sizeof(myCtx->debugf_filename))
                                                  != OCI_SUCCESS) ||
        (OCIExtractToStr(envhp, errhp, "debugf_path",
                         0, myCtx->debugf_path,
                         sizeof(myCtx->debugf_path))
                                                  != OCI_SUCCESS) ||
        (OCIExtractTerm(envhp, errhp) != OCI_SUCCESS))
    {
        raise_application_error(myCtx, 20000, "%s",
                                lastOciError(myCtx));
        return NULL;
    }
}
```

Далее следует блок кода, который будет выполняться при втором и последующих вызовах функции init в сеансе. Поскольку функция OCIContextGetValue для второго и последующих вызовов возвращает контекст, мы просто устанавливаем на него соответствующие указатели в структуре:

```
else
{
    myCtx->ctx   = ctx;
    myCtx->envhp = envhp;
    myCtx->svchp = svchp;
    myCtx->errhp = errhp;
}
```

Последнее действие в функции **init** — вызов функции **OCIFileInit.** Она инициализирует набор функций OCI для работы с файлами, чтобы можно было открывать файлы операционной системы для чтения/записи. Можно было бы использовать и стандартные функции **fopen, fclose, fread** и **fwrite** языка C. Использованный подход позволяет сделать внешнюю процедуру более переносимой и обеспечить единообразную обработку ошибок на различных платформах. **В** функцию **init** можно добавить и другие вызовы. Например, если предполагается использование функций **OCIFormat*** (аналогичных функции **vsprintf** языка C), можно добавить в функцию инициализации вызов **OCIFormatInit.** Не забудьте при этом добавить соответствующий вызов **OCIFormatTerm** в представленную ниже функцию **term.**

```
    if (OCIFileInit(myCtx->envhp, myCtx->errhp) != OCI_SUCCESS)
    {
        raise_application_error(myCtx, 20000, "%s", lastOciError(myCtx));
        return NULL;
    }
    return myCtx;
)
```

Теперь перейдем к упомянутой функции term. Это функция завершения и очистки; ее нужно вызывать после каждого успешного вызова функции init. Это должна быть последняя функция, вызываемая перед возвратом управления из внешней процедуры на языке C в SQL:

```
static void term(myCtxStruct * myCtx)
{
    OCIFileTerm(myCtx->envhp, myCtx->errhp);
}
```

Создание шаблона закончено. Я использую один и тот же шаблон исходного кода для всех своих проектов по созданию внешних процедур (с небольшими изменениями, если используется только библиотека OCI, а не сочетание Pro*C и вызовов OCI). Такой шаблон экономит большое количество времени и обеспечивает множество функциональных возможностей.

Теперь начнем добавлять специфический код. Сразу же после общих компонентов я перечисляю все коды ошибок, которые будет возвращать функция, начиная с 20001. Перечислив их в самом начале, можно задать соответствующие исключительные ситуации с помощью конструкций pragma exception_init в коде на языке PL/SQL. Это позволит перехватывать в программах на PL/SQL исключительные ситуации с определенными именами, а не проверять коды ошибок. Я не буду демонстрировать это в данном примере, но в следующем примере с использованием прекомпилятора Pro*C — продемонстрирую. Коды ошибок должны быть в диапазоне от 20000 до 20999, поскольку именно эти коды ошибок выделяются сервером Oracle для приложений; остальные коды ошибок ИСПОЛЬЗУЮТСЯ самим сервером.

```
#define  ERROR_OCI_SRROR  20001
#define ERROR_STR_TOO_SMALL 20002
#define ERROR_RAW_TOO_SMALL 20003
#define ERROR_CLOB_NULL    20004
#define  ERROR_ARRAY_NULL   20005
```

Переходим к первой специфической функции. Это реализация процедуры **pass_number,** заданной в представленном ранее PL/SQL-пакете. Она принимает из PL/SQL параметр типа **NUMBER** в режиме IN и устанавливает параметр типа **NUMBER,** переданный в режиме **OUT.** Функция выполняет следующие действия.

- Использует внутренний тип данных Oracle **OCINumber** с помощью соответствующих функций. В данном случае тип данных Oracle NUMBER преобразуется в тип данных **double** языка C с помощью встроенной функции **OCINumberToReal.** Можно преобразовать данные типа NUMBER в строку с помощью функции **OCINumberToText** или в тип данных **int** языка C с помощью функции **OCINumberToInt. В** библиотеке **OCI** имеется почти SO числовых функций для выполнения различных операций с внутренним типом данных. Описание всех имеющихся функций можно найти в руководстве *Oracle Call Interface Programmer's Guide.*
- Обрабатывает данные типа **double.** В данном случае мы просто меняем знак числа: если число было положительным, мы делаем его отрицательным, и наоборот.
- Устанавливает параметру типа **NUMBER,** переданному в режиме **OUT,** полученное значение с измененным знаком, и завершает работу.

Перед каждой функцией, вызываемой из языка PL/SQL, указывается макрос, обеспечивающий ее переносимость. Этот макрос "экспортирует" функцию. Это необходимо только на платформе Windows, а в ОС UNIX — нет. Я обычно включаю этот макрос независимо от платформы, для которой создается внешняя процедура, поскольку мне часто приходится переносить библиотеки внешних процедур из Windows в UNIX, и наоборот. Постоянное включение макроса упрощает перенос. Встроенные в код комментарии объясняют его назначение по ходу дела:

```
#ifdef WIN_NT
_declspec(dllexport)
#endif
void pass_number
  (OCIExtProcContext * ctx      /* контекст */,

   OCINumber *       p_inum     /* OCINumber */,
   short             p_inum_i   /* INDICATOR short */,

   OCINumber *       p_onum     /* OCINumber */,
   short *           p_onum_i   /* INDICATOR short */)
{
double    l_inum;
myCtxStruct*myCtx;
```

До выполнения **любых действий** необходимо получить контекст сеанса. При этом будет получена среда OCI, значения параметров и т.д. Этот вызов будет выполняться первым во всех внешних процедурах:

```
if ((myCtx = init( ctx )) = NULL) return;
debugf(myCtx, "Входим в функцию Pass Number");
```

Обратимся к первому параметру. Мы передали его как данные типа OCINumber. Теперь к нему можно применять множество функций OCINumber*. В данном случае

данные типа **NUMBER** преобразуются в данные типа **double** языка C с помощью функции **OCINumberToReal.** Так же просто преобразовать их в данные типа **int, long, float** или в форматированную строку.

Сначала необходимо убедиться, что передано **не пустое** значение типа **NUMBER;** если — да, оно обрабатывается, если же — нет, вызывается функция **term().** Если удалось успешно получить первый параметр, мы изменяем его знак, после чего создаем на основе полученного значения данные типа **OCINumber** с помощью функции **OCINumberFromReal.** Если это получилось, устанавливаем в индикаторе пустого значения **p_onum_I** признак **NOTNULL,** чтобы в вызывающей среде можно было понять, что возвращается непустое значение. Обработка закончена. Вызываем функцию **term** для освобождения ресурсов и возвращаем управление:

```
    if (p_inum_i = OCI_IND_NOTNULL)
    {
       if (OCINumberToReal(myCtx->errhp, p_inum, sizeof(l_inum), &l_inum)
                 !=OCI_SUCCESS)
       {
            raise_application_error(myCtx,ERROR_OCI_ERROR,
                                    "%S",lastOciError(myCtx));
       }
       else
       {
           debugf(myCtx, "Первый параметр: %g", l_inum);
           l_inum = -l_inum;
           if (OCINumberFromReal(myCtx->errhp, &l_inum,
                                 sizeof(l_inum), p_onum) !=OCI_SUCCESS)
           {
                raise_application_error(myCtx,ERROR_OCI_ERROR,
                                        "%s",lastOciError(myCtx));
           }
           else
           {
               *p_onum_i = OCI_IND_NOTNULL;
               debugf(myCtx,
                     "Устанавливаем параметр OUT равным %g, а индикаторную
->переменную—равной NOTNULL",
                           l_inum);
           }
        }
    }
    term(myCtx);
}
```

Вот и все. Наша первая внешняя процедура использует все вспомогательные функции: **raise_application_error**, **lastOciError**, init, **term** и **debugf**. При тестировании этой процедуры мы проверим результаты вызова функции **debugf**. Они подтвердят, что функция делает именно то, что и предполагалось (представляя собой удобное средство для отладки).

Обратите внимание, как я позаботился о возвращении управления из данной функции только в одном месте. Если возврат управления происходит в нескольких местах, не забудьте в каждом из них вызывать функцию term(myCtx).

Теперь переходим к остальным функциям. Следующая функция обрабатывает даты, переданные как параметры в режиме **IN** и **OUT**. Мы будем:

- принимать параметр типа **DATE** в режиме **IN**;
- форматировать его как строку с помощью соответствующих функций библиотеки OCI для трассировки (для работы с данными типа DATE предлагается около 16 функций OCI);
- добавлять один месяц к дате с помощью соответствующей функции OCI;
- присваивать новую дату параметру, переданному в режиме OUT;
- преобразовывать только что присвоенную дату в строку и выдавать ее;
- наконец, вызывать функцию term и возвращать управление.

```
#ifdef WIN_NT
_declspec (dllexport)
#endif
void pass_date
  (OCIExtProcContext * ctx        /* CONTEXT */,
  OCIDate *          p_idate    /* OCIDATE */,
  short              p_idate_i  /* INDICATOR short */,
  OCIDate *          p_odate    /* OCIDATE */,
  short *            p_odate_i  /* INDICATOR short */
 )
{
char      buffer[255];
ub4       buff_len;
char      * fmt = "dd-mon-yyyy hh24:mi:ss";
myCtxStruct*myCtx;
    if ((myCtx = init( ctx )) = NULL) return;
    debugf(myCtx, "Входим в функцию Pass Date");
    if  (p_idate_i  = OCI_IND_NOTNULL)
    {
        buff_len = sizeof(buffer);
        if (OCIDateToText(myCtx->errhp, p_idate, fmt, strlen(fmt),
                          NULL, -1, &buff_len, buffer) != OCI_SUCCESS)
        {
            raise_application_error(myCtx,ERROR_OCI_ERROR,
                                    "%s", lastOciError(myCtx));
        }
        else
        {
            debugf(myCtx, "Входной параметр типа date имел значение '%.*s'",
                   buff_len, buffer);
            if (OCIDateAddMonths(myCtx->errhp, p_idate, 1, p_odate)
                    != OCI_SUCCESS)
```

```
            {
                raise_application_error(myCtx,ERROR_OCI_ERROR,
                                        "%s",lastOciError(myCtx));
            }
            else
            {
                *p_odate_i = OCI_IND_NOTNULL;
                buff_len = sizeof(buffer);
                if (OCIDateToText(myCtx->errhp, p_odate, fmt,
                                  strlen(fmt), NULL, -1,
                                  &buff_len, buffer) != OCI_SUCCESS)
                {
                    raise_application_error(myCtx,ERROR_OCI_ERROR,
                                            "%s",lastOciError(myCtx));
                )
                else
                {
                    debugf(myCtx,
                   "Выходной параметр типа date получил значение '%.*s'",
                            buff_len, buffer);
                }
            }
        }
    }
    term(myCtx);}
```

Теперь разберемся, что необходимо для приема и передачи строк. Работать со строками несколько проще, чем с данными типа **NUMBER** и **DATE**, поскольку они передаются просто как строки **ASCII**, завершаемые нулем. Для всех строк, передаваемых в режиме **OUT**, будет использоваться параметр **MAXLEN**. Параметр **MAXLEN** задает для возвращаемой строки максимальный размер буфера, который может изменяться при каждом вызове. Дело в том, что буфер предоставляет вызывающая среда, и при каждом вызове может передаваться в **режиме OUT** другой параметр другого размера. В результате внешняя процедура сможет учесть размер и предотвратить переполнение буфера. Она проинформирует вызывающую среду о том, что предоставлен слишком маленький буфер, и сообщит, какого размера он должен быть.

```
#ifdef WIN_NT
_declspec(dllexport)
#endif
voidpass_str
  (OCIExtProcContext * ctx       /* CONTEXT */,

   char *            p_istr     /* STRING */,
   short             p_istr_i   /* INDICATOR short */,
   char *            p_ostr     /* STRING */,
   short *           p_ostr_i   /* INDICATOR short */,
   int *             p_ostr_ml  /* MAXLEN int */
  )
{
myCtxStruct*myCtx;

    if ((myCtx = init(ctx)) = NULL) return;
```

```
    debugf(myCtx, "Входим в функцию Pass Str");

    if (p_istr_i = OCI_IND_NOTNULL)
    {
    int     l_istr_l  = strlen(p_istr);

        if (*p_ostr_ml > l_istr_l)
        {

            strcpy(p_ostr,p_istr);
            strupr(p_ostr);
            *p_ostr_i = OCI_IND_NOTNULL;
        >
        else
        {
            raise_application_error(myCtx, ERROR_STR_TOO_SMALL,
               "выходной буфер размером %d байт должен быть длиной не
->менее%d байт",
                *p_ostr_ml, l_istr_l+1);
        }
    }
    term(myCtx);
}
```

На примере следующей функции я продемонстрирую использование типа binary_integer. Тип binary_integer в PL/SQL представляет 32-битовое целое число со знаком. Передать его проще всего. Значения этого типа передаются естественным для программиста на языке C образом. Эта функция просто проверит входное значение и присвоит его (умноженное на 10) выходному параметру:

```
#ifdef WIN_NT
_declspec (dllexport)
#endif
void pass_int
 (OCIExtProcContext * ctx        /* CONTEXT */,

   int               p_iINT     /* int */,
   short             p_iINT_i   /* INDICATOR short */,

   int *          P_oINT        /* int */,
   short *           p_oINT_i   /* INDICATOR short */
 )
{
myCtxStruct*myCtx;

    if ((myCtx = init(ctx)) == NULL ) return;
    debugf(myCtx, "Входим в функцию Pass Int");

    if (p_iINT_i == OCI_IND_NOTNULL)
    <
        debugf(myCtx, "Первый параметр типа INT имеет значение %d", p_iINT);

        *p_oINT = p_iINT*10;
        *p_oINT_i = OCI_IND_NOTNULL;

        debugf(myCtx, "Устанавливаем выходному параметру типа INT
->значение%d", *p_oINT);
```

```
    term(myCtx);
}
```

Теперь рассмотрим передачу параметров PL/SQL типа **BOOLEAN**. Тип **BOOLEAN** языка PL/SQL в данном случае сопоставляется типу int языка C. Значение 1 представляет истину, а значение 0 — ложь, как и следовало ожидать. Эта функция просто проверяет входной параметр (не пуст ли он) и устанавливает выходной параметр равным отрицанию входного. И в этом случае, поскольку обеспечивается простое сопоставление со встроенным типом языка C, реализовать эту функцию очень легко. Для обмена данными не надо использовать дескрипторы среды или вызовы функций. Функция просто устанавливает выходной параметр равным отрицанию входного:

```
#ifdef WIN_NT
_declspec(dllexport)
#endif
voidpass_bool
  (OCIExtProcContext * ctx       /* CONTEXT */,

   int               p_ibool     /* int */,
   short             p_ibool_i   /* INDICATOR short */,

   int *             p_obool     /* int */,
   short *           p_obool_i   /* INDICATOR short */)
{
myCtxStruct*myCtx;

    if ((myCtx = init(ctx)) = NULL) return;
    debugf(myCtx, "Входим в функцию Pass Boolean");

    if (p_ibool_i = OCI_IND_NOTNULL)
    {
        *p_obool = !p_ibool;
        *p_obool_i = OCI_IND_NOTNULL;
    }
    term(myCtx);
}
```

Теперь займемся передачей параметров типа **RAW.** Поскольку переменные типа **VARCHAR2** в PL/SQL могут иметь длину не более 32 Кбайт, мы всегда будем использовать более простой для взаимодействия внешний тип данных **RAW.** Он соответствуете языке C типу данных **unsigned char** *, который представляет собой указатель на байты данных. Работая с данными типа **RAW,** мы **всегда** будем получать атрибут **LENGTH.** Это обязательно, поскольку нет другого способа определить, какое количество данных надо обрабатывать. Мы также **всегда** будем получать атрибут **MAXLEN** для всех параметров, переданных в режиме OUT и имеющих переменную длину, чтобы избежать потенциальной перезаписи буфера. Этот атрибут, хотя технически и не обязателен, слишком важен, чтобы его не использовать. Данная функция просто копирует входной буфере выходной:

```
#ifdef WIN_NT
_declspec(dllexport)
#endif
```

```
void pass_raw
  (OCIExtProcContext * ctx     /* CONTEXT */,

   unsigned char *  p_iraw     /* RAW */,
   short            p_iraw_i   /* INDICATOR short */,
   int              p_iraw_l   /* LENGTH INT */,

   unsigned char *  p_oraw     /* RAW */,
   short *          p_oraw_i   /* INDICATOR short */,
   int *            p_oraw_ml  /* MAXLEN int */,
   int *            p_oraw_l   /* LENGTH int */
 )
{
myCtxStruct*myCtx;

    if ((myCtx - init(ctx)) == NULL ) return;
    debugf(myCtx, "Входим в функцию Pass long raw");
    if  (p_iraw_i  = OCI_IND_NOTNULL)
    {
        if  (p_iraw_l <= *p_oraw_ml)
        {
            memcpy(p_oraw, p_iraw, p_iraw_l);

            *p_oraw_l = p_iraw_l;
            *p_oraw_i = OCI_IND_NOTNULL;
        }
        else
        {
            raise_application_error(myCtx, ERROR_RAW_TOO_SMALL,
          "Буфер размером %d байт должен быть размером не менее %d байт",
                      *p_oraw_ml, p_iraw_l);
        }
    }
    else
    {
        *p_oraw_i = OCI_IND_NULL;
        *p_oraw_l =  0;
    }
    term(myCtx);
}
```

Последняя функция, обрабатывающая скалярные данные, работает с большими объектами. Работать с большими объектами ничуть не сложнее, чем с данными типа **DATE** или **NUMBER.** Имеется несколько функций библиотеки OCI, позволяющих читать и записывать большие объекты, копировать и сравнивать их, и т.д. В этом примере используются функции для определения длины и последующего копирования большого входного объекта в выходной. Эта функция требует, чтобы в вызывающей среде был проинициализирован большой объект, передаваемый в режиме **OUT** (либо путем выбора локатора **LOB** из существующей строки таблицы, либо с помощью подпрограммы **dbms_lob.createtemporary**). Хотя я демонстрирую работу только с большим объектом типа **CLOB,** реализации для объектов типа **BLOB** и **BFILE** будут очень похожи: для всех трех

типов используется тип данных **OCILobLocator.** Подробнее о функциях, работающих с данными типа **OCILobLocator,** можно прочитать в руководстве *Oracle Call Interface Programmer's Guide*. Приведенная функция просто копирует входной объект типа **CLOB** в выходной.

```
#ifdef WIN_NT
_declspec (dllexport)
#endif
void pass_clob
  (OCIExtProcContext * ctx        /* CONTEXT */,
   OCILobLocator *    p_iCLOB    /* OCILOBLOCATOR */,
   short              p_iCLOB_i  /* INDICATOR short */,
   OCILobLocator * *  p_oCLOB    /* OCILOBLOCATOR */,
   short *            p_oCLOB_i  /* INDICATOR short */
  )
{
ub4          lob_length;
myCtxStruct* myCtx;
    if ((myCtx = init(ctx)) = NULL ) return;
    debugf(myCtx, "Входим в функцию Pass Clob" );
    if (p_iCLOB_i = OCI_IND_NOTNULL && *p_oCLOB_i == OCI_IND_NOTNULL)
    {
        debugf(myCtx, "оба больших объекта - NOT NULL");
        if (OCILobGetLength(myCtx->svchp, myCtx->errhp,
                            p_iCLOB, Slob_length) != OCI_SUCCESS)
        {
            raise_application_error(myCtx,ERROR_OCI_ERROR,
                                    "%s",lastOciError(myCtx));
        }
        else
        {
            debugf(myCtx, "Длина входного большого объекта была %d",
                                                          lob_length);
            if (OCILobCopy(myCtx->svchp, myCtx->errhp, *p_oCLOB, p_iCLOB,
                           lob_length, 1, 1) != OCI_SUCCESS)
            {
                raise_application_error(myCtx,ERROR_OCI_ERROR,
                                        "%s",lastOciError(myCtx));
            }
            else
            {
                debugf(myCtx, "Мы скопировали большой объект!");
            }
        }
    }
    else
    {
        raise_application_error(myCtx, ERROR_CLOB_NULL,
                                "%s %s объект типа clob был пустым (NULL)",
                               (p_iCLOB_i = OCI_IND_NULL)?"входной":"",
```

```
                                        (*p_oCLOB_i=OCI_IND_NULL)?"выходной":"");
    }
    term(myCtx);
}
```

Следующие три функции демонстрируют, как передавать и принимать массивы данных во внешней процедуре. Если помните, мы создали несколько табличных типов в SQL: **numArray, dateArray** и **strArray.** Эти типы будут использоваться для демонстрации. Функции будут показывать, сколько элементов передано в массиве, выдавать их значения и наполнять этими элементами массив, переданный в режиме OUT.

В этих функциях, работающих с массивами, мы будем использовать набор функций **OCIColl*.** Для работы с наборами (массивами) можно использовать около 15 функций, позволяющих выполнять итерацию по элементам, получать или устанавливать значения элементов и т.п. Ниже использованы следующие наиболее типичные функции:

- **OCICollSize,** для получения количества элементов в массиве;
- **OCICollGetElem,** для получения i-го элемента массива;
- **OCICollAppend,** для добавления элемента в конец массива.

Полный список имеющихся функций можно найти в руководстве *Oracle Call Interface Programmer's Guide*.

Начнем с массива чисел. Эта функция будет проходить по всем элементам входного набора, выдавать их значения и присваивать соответствующим элементам выходного набора:

```
#ifdef WIN_NT
_declspec (dllexport)
#endif
void pass_numArray
 (OCIExtProcContext * ctx          /* CONTEXT */,
  OCIColl *           p_in         /* OCICOL  */,
  short               P_in_i       /* INDICATOR short */,
  OCIColl **          p_out        /* OCICOL  */,
  short *             p_out_i      /* INDICATOR short */
 )
{
ub4          arraySize;
double       tmp_dbl;
boolean      exists;
OCINumber *ocinum;
int          i;
myCtxStruct*myCtx;

    if ((myCtx = init(ctx)) == NULL) return;
    debugf(myCtx, "Входим в функцию Pass numArray");

    if (p_in_i == OCI_IND_NULL)
    {
        raise_application_error(myCtx, ERROR_ARRAY_NULL,
                                        "Входной массив – NULL");
    }
```

```
    else
    if (OCICollSize(myCtx->envhp, myCtx->errhp,
                    p_in, &arraySize) != OCI_SUCCESS)
    {
        raise_application_error(myCtx,ERROR_OCI_ERROR,
                                "%s",lastOciError(myCtx));
}
else
{
        debugf(myCtx, "Входной массив состоит из %d элементов", arraySize);
        for(i - 0; i < arraySize; i++)
        {
            if (OCICollGetElem(myCtx->envhp, myCtx->errhp, p_in, i,
                           Sexists, (dvoid*)&ocinum, 0) != OCI_SUCCESS)
            {
                raise_application_error(myCtx,ERROR_OCI_ERROR,
                                        "%s",lastOciError(myCtx));
                break;
            }
            if (OCINumberToReal( myCtx->errhp, ocinum,
                             sizeof(tmp_dbl), &tmp_dbl) != OCI_SUCCESS)
            {
                raise_application_error(myCtx,ERROR_OCI_ERROR,"%s",
                                        lastOciError(myCtx));
                break;
            }
            debugf(myCtx, "p_in[%d] = %g", i, tmp_dbl);
            if (OCICollAppend(myCtx->envhp, myCtx->errhp, ocinum, 0,
                              *p_out) != OCI_SUCCESS )
            {
                raise_application_error(myCtx,ERROR_OCI_ERROR,
                                        "%s",lastOciError(myCtx));
                break;
            }
            debugf(myCtx, "Элемент добавлен в конец другого массива");
        }
        *p_out_i = OCI_IND_NOTNULL;
    }
    term(myCtx);
}
```

Следующие две функции — для массивов строк и дат. Они очень похожи на представленную выше функцию, работающую с массивами чисел, поскольку все три функции работают с данными типа OCIColl *. Пример для типа данных strArray интересен тем, что в нем впервые используется новый тип данных библиотеки OCI — OCIString (это не то же самое, что тип char *). При использовании типа данных OCIString необходимо работать со ссылками на ссылки. Для строк и дат мы будем выполнять те же действия, что и в представленном ранее примере для чисел:

```
#ifdef WIN_NT
_declspec (dllexport)
```

```
#endif
void pass_strArray
  (OCIExtProcContext * ctx        /* CONTEXT */,
   OCIColl *           p_in       /* OCICOL  */,
   short               P_in_i     /* INDICATOR short */,
   OCIColl **          p_out      /* OCICOL  */,
   short *             p_out_i    /* INDICATOR short */
  )
{
ub4        arraySize;
boolean    exists;
OCIString * * ocistring;
int        i;
text      *txt;
myCtxStruct*myCtx;

    if ((myCtx = init(ctx)) — NULL) return;
    debugf( myCtx,  "Входим в функцию Pass strArray");

    if (p_in_i = OCI_IND_NULL)
    {
        raise_application_error(myCtx, ERROR_ARRAY_NULL,
                                    "Входной массив — NULL");
    }
    else if (OCICollSize( myCtx->envhp, myCtx->errhp,
                          p_in, &arraySize) != OCI_SUCCESS)
    {
        raise_application_error(myCtx,ERROR_OCI_ERROR,
                               "%s",lastOciError(myCtx));
    }
    else
    {
        debugf(myCtx, "Входной массив состоит из %d элементов", arraySize);
        for(i =0; i < arraySize; i++)
        {
           if (OCICollGetElem(myCtx->envhp, myCtx->errhp, p_in, i,
                        &exists, (dvoid*)socistring, 0) != OCI_SUCCESS)
            {
                raise_application_error(myCtx,ERROR_OCI_ERROR,
                                    "%s",lastOciError(myCtx));
                break;
            }
            txt = OCIStringPtr(myCtx->envhp, *ocistring);
            debugf( myCtx,  "p_in[%d] = 's', size = %d, exists = %d",
                i, txt, OCIStringSize(myCtx->envhp,*ocistring), exists);
            if (OCICollAppend(myCtx->envhp,myCtx->errhp, *ocistring,
                              0, *p_out) != OCI_SUCCESS)
            {
                raise_application_error(myCtx,ERROR_OCI_ERROR,
                                    "%s",lastOciError(myCtx));
                break;
            }
```

```
            debugf(myCtx, "Элемент добавлен в конец другого массива");
        }
        *p_out_i = OCI_IND_NOTNULL;
    }
    term(myCtx);
}
#ifdef WIN_NT
_declspec (dllexport)
#endif
void pass_dateArray
  (OCIExtProcContext *     ctx     /* CONTEXT */,
   OCIColl *           p_in       /* OCICOL  */,
   short          P_in_i          /* INDICATOR short */,
   OCIColl **          p_out      /* OCICOL  */,
   short *             p_out_i    /* INDICATOR short */
  )
{
ub4        arraySize;
boolean    exists;
OCIDate *  ocidate;
int        i;
char     * fmt = "Day, Month YYYY hh24:mi:ss";
ub4        buff_len;
char       buffer[255];
myCtxStruct*myCtx;
    if ((myCtx = init(ctx)) = NULL) return;
    debugf(myCtx, "Входим в функцию Pass dateArray");
    if (p_in_i == OCI_IND_NULL)
    {
        raise_application_error(myCtx, ERROR_ARRAY_NULL,
                                        "Входной массив — NULL");
    }
    else if (OCICollSize(myCtx->envhp, myCtx->errhp,
                          p_in, &arraySize) != OCI_SUCCESS)
    {
        raise_application_error(myCtx,ERROR_OCI_ERROR,
                                "%s",lastOciError(myCtx));
    }
    else
    {
        debugf(myCtx, "Входной массив состоит из %d элементов", arraySize);
        for(i = 0 ; i < arraySize; i++)
        {
            if (OCICollGetElem( myCtx->envhp, myCtx->errhp, p_in, i,
                          &exists, (dvoid*)&ocidate, 0) != OCI_SUCCESS)
            {
                raise_application_error(myCtx,ERROR_OCI_ERROR,
                                        "%s",lastOciError(myCtx));
```

```
                break;
            }
            buff_len = sizeof(buffer);
            if (OCIDateToText(myCtx->errhp, ocidate, fmt, strlen(fmt),
                              NULL, -1, &buff_len, buffer) != OCI_SUCCESS)
            {
                raise_application_error(myCtx,ERROR_OCI_ERROR,
                                        "%s",lastOciError(myCtx));
                break;
            }
            debugf(myCtx, "p_in[%d] = %.*s", i, buff_len, buffer);
            if (OCICollAppend(myCtx->envhp,myCtx->errhp, ocidate,
                              0, *p_out ) != OCI_SUCCESS)
            {
                raise_application_error(myCtx,ERROR_OCI_ERROR,
                                        "%s",lastOciError(myCtx));
                break;
            )
            debugf(myCtx, "Элемент добавлен в конец другого массива");
        }
        *p_out_i = OCI_IND_NOTNULL;
    }
    term(myCtx);
}
```

В завершение рассмотрим функции, непосредственно возвращающие значения. Они выглядят немного необычно, поскольку в PL/SQL используются функции без параметров, возвращающие значения, но соответствующие функции на языке C **должны** принимать ряд параметров. Другими словами, простейшей функции PL/SQL без параметров будет соответствовать C-функция с формальными параметрами. Эти формальные параметры будут использоваться внешней процедурой для передачи серверу Oracle следующих данных:

- индикаторной переменной, показывающей, вернула ли функция значение **NULL;**
- текущей длины данных строкового типа или типа **RAW.**

С этими параметрами мы уже встречались, просто странно, что они передаются функции.

```
#ifdef WIN_NT
_declspec (dllexport)
#endif
OCINumber * return_number
  (OCIExtProcContext * ctx,
   short *             return_i)
{
double       our_number = 123.456;
OCINumber *  return_value;
myCtxStruct*myCtx;
    *return_i = OCI_IND_NULL;
```

```
if ((myCtx = init(ctx)) = NULL) return NULL;
debugf(myCtx, "Входим в функцию, возвращающую Number");
```

Здесь необходимо выделить память для возвращаемого числа. Нельзя использовать стековую переменную, поскольку при возврате значения она выходит из области действия. При выделении памяти с помощью функции **malloc** произойдет утечка памяти. Использовать статическую переменную тоже нельзя, поскольку из-за кэширования внешних процедур другой сеанс может изменить значение, на которое мы сослались, после его возврата (но до того, как сервер Oracle его скопирует). Единственно корректный способ — выделить память следующим образом:

```
return_value =
      (OCINumber *)OCIExtProcAllocCallMemory(ctx, sizeof(OCINuntoer));
if(return_value == NULL)
{
    raise_application_error(myCtx, ERROR_OCI_ERROR,"%s",
                                          "не хватает памяти");
}
else
{
    if (OCINumberFromReal(myCtx->errhp, &our_number,
                sizeof(our_number), return_value) != OCI_SUCCESS)
    {
        raise_application_error(myCtx,ERROR_OCI_ERROR,
                                "%s",lastOciError(myCtx));
    }
    *return_i = OCI_IND_NOTNULL;
}
term(myCtx);
return return_value;
}
```

Возврат даты очень похож на возврат числа. Возникают те же проблемы с памятью. Выделяется память для возвращаемого значения типа DATE, в нее записывается значение, устанавливается значение индикаторной переменной и возвращается результат:

```
#ifdef WIN_NT
_declspec (dllexport)
#endif
OCIDate * return_date
  (OCIExtProcContext * ctx,
   short *             return_i)
{
OCIDate * return_value;
myCtxStruct*myCtx;

    if ((myCtx = init(ctx)) == NULL) return NULL;
    debugf(myCtx, "Входим в функцию, возвращающую данные типа Date">;

    return_value =
          (OCIDate *)OCIExtProcAllocCallMemory(ctx, sizeof(OCIDate));

    if (return_value == NULL)
```

```
        raise_application_error(myCtx, ERROR_OCI_ERROR, "%s",
                                                "не хватает памяти");
    }
    else
    {
        *return_i = OCI_IND_NULL;
        if (OCIDateSysDate(myCtx->errhp, return_value) != OCI_SUCCESS)
        {
            raise_application_error(myCtx,ERROR_OCI_ERROR,
                                    "%s",lastOciError(myCtx));
        }
        *return_i = OCI_IND_NOTNOLL;
    }
    term(myCtx);
    return return_value;
}
```

При возврате строковых данных (**VARCHAR**) будут использоваться два параметра: индикаторная переменная и поле **LENGTH**. В этом случае, как и для параметра, переданного в режиме **OUT**, задается поле **LENGTH**, чтобы в вызывающей среде была известна длина возвращенной строки.

Представленные выше соображения во многом применимы и при возврате строк: выделяется память, устанавливается индикаторная переменная, задается и возвращается значение:

```
#ifdef WIN_NT
_declspec (dllexport)
#endif
char * return_string
 (OCIExtProcContext * ctx,
  short *             return_i,
  int   *             return_l)
{
char * data_we_want_to_return - "Hello World!";
char * return_value;
myCtxStruct*myCtx;
    if ((myCtx = init(ctx)) — NOLL) return NULL;
    debugf(myCtx, "Входим в функцию, возвращающую строку " ) ;
    return_value = (char *)OCIExtProcAllocCallMemory(ctx,
                                    strlen(data_we_want_to_return)+1);
    if(return_value = NULL)
  {
      raise_application_error(myCtx, ERROR_OCI_ERROR, "%s",
                                                "не хватает памяти");
    }
    else
    {
        *return_i = OCI_IND_NULL;
        strcpy(return_value, data_we_want_to_return);
        *return_l = strlen(return_value);
```

```
            *return_i = OCI_IND_NOTNULL;
        }
        term(myCtx);
        return return_value;
    }
```

Мы рассмотрели код на языке C, необходимый для демонстрации способов передачи всех основных типов данных в режиме **IN** и **IN/OUT,** а также возврата данных соответствующих типов из функций. Было представлено также множество функций библиотеки OCI, использующихся во внешних процедурах, в частности функции для создания и получения контекста с целью поддержки информации о состоянии, функции для обработки файлов параметров, создания и записи файлов ОС. Не продемонстрировано следующее.

- Передача и получение данных **сложных** объектных типов во внешних процедурах. Делается это примерно так же, как и в примерах с массивами (поскольку там передавались и принимались данные простых объектных типов). Для работы с входными и выходными данными объектных типов используются предоставляемые библиотекой OCI средства работы с компонентами объекта.
- Возврат всех необходимых типов из функций. Я представил только возврат строк, дат и чисел. Возврат данных остальных типов выполняется аналогично (несколько проще для данных типа **int,** поскольку при этом не надо выделять память).

Сейчас мы рассмотрим файлы управления проектом, **make**-файлы, которые можно использовать для создания внешних процедур в среде ОС UNIX или Windows.

Создание внешней процедуры

Давайте сначала рассмотрим универсальный make-файл для Windows:

```
CPU=i386

MSDEV       = c:\msdev                                          (1)

ORACLE_HOME = c:\oracle                                         (2)

!include <$(MSDEV)\include\win32.mak>                           (3)

TGTDLL  = extproc.dll                                           (4)
OBJS    = extproc.obj                                           (5)
NTUSER32LIBS    = $(MSDEV)\lib\user32.1ib   \                   (6)
                  $(MSDEV)\lib\msvcrt.lib  \
                  $(MSDEV)\lib\oldnames.lib  \
                  $(MSDEV)\lib\kernel32.1ib   \
                  $(MSDEV)\lib\advapi32.lib

SQLLIB  = $(ORACLE_HOME)\precomp\lib\msvc\orasql8.1ib  \        (7)
          $(ORACLE_HOME)\oci\lib\msvc\oci.lib

INCLS   = -I$(MSDEV)\include \                                  (8)
          -I$(ORACLE_HOME)\oci\include \
          -I.

CFLAGS  = $(INCLS) -DWIN32 -DWIN_NT -D_DLL                      (9)
```

```
all: $(TGTDLL)                                                    (10)

clean:                                                            (11)
    erase *.obj *.lib *.exp

$(TGTDLL): $(OBJS)                                                (12)
    $(link) -DLL $(dllflags) \
        /NODEFAULTLIB:LIBC.LIB -out:$(TGTDLL) \
        $(0BJS) \
        $(NTUSER32LIBS) \
        $(SQLLIB)
```

*Выделенные **полужирным** числа в круглых скобках не являются частью файла управления проектом, они указаны лишь для возможности ссылки на них в дальнейшем.*

1. Это каталог, в котором у меня установлен компилятор языка C. Я использую компилятор Microsoft Visual C/C++, который поддерживается в среде Windows. Значение этой переменной я использую позже в make-файле, когда необходимо сослаться на этот каталог.
2. Мой каталог **ORACLE_HOME.** Он используется при поиске включаемых файлов для OCI/Pro*C и стандартных библиотек Oracle.
3. Я включаю стандартный шаблон make-файла Microsoft. Он задает переменные **$(link)** и **$(dllflags),** которые могут быть разными для различных версий компилятора.
4. Переменная **TGTDLL** задает имя создаваемой DLL-библиотеки.
5. Переменная **OBJS** задает список объектных файлов, используемых при создании библиотеки. Если распределить код по нескольким файлам, в списке будет указано несколько объектных файлов. В нашем несложном примере используется только один объектный файл.
6. Переменная **NTUSER32LIBS** содержит список стандартных системных библиотек, с которыми выполняется компоновка.
7. Переменная **SQLLIB** содержит список необходимых библиотек Oracle. В данном примере я компоную библиотеки **как Pro***C, **так и** OCI, хотя используются только библиотеки OCI. Но от включения библиотек Pro*C вреда не будет.
8. Переменная **INCLS** содержит список каталогов, в которых находятся необходимые включаемые файлы. В данном случае мне необходимы системные заголовочные файлы, а также заголовочные файлы сервера Oracle и текущий рабочий каталог.
9. Переменная **CFLAGS** — стандартный макрос языка C, используемый компилятором. Я определяю макрос **-DWIN_NT,** для условной компиляции кода, предназначенного для NT (например, **_declspec(dllexport)).**
10. Цель **all:** по умолчанию будет создавать DLL-библиотеку.
11. Цель **clean:** требует удалить временные файлы, созданные в ходе компиляции.
12. Цель **TGTDLL** требует выполнить команду, создающую DLL-библиотеку. Она скомпилирует и скомпонует весь необходимый код.

Как разработчик я постоянно использую этот make-файл. Обычно я изменяю только строку (4) — имя библиотеки и строку (5) — список объектных файлов. Остальные компоненты make-файла изменять после первоначального конфигурирования не придется.

Выполнив команду nmake, мы увидим примерно следующее:

```
C:\Documents and Settings\Thomas Kyte\Desktop\extproc\demo_passing>nmake

Microsoft (R) Program Maintenance Utility  Version 1.60.5270
Copyright (c) Microsoft Corp 1988-1995. All rights reserved.
        cl -Ic:\msdev\include  -Ic:\oracle\oci\include  -I. -DWIN32
->-DWIN_NT -D_DLL /c extproc.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 10.00.5270 for
->80x86
Copyright (C) Microsoft Corp 1984-1995. All rights reserved.

extproc.c
        link -DLL   /NODEFAULTLIB:LIBC.LIB -out:extproc.dll  extproc.obj
c:\msdev\lib\user32.lib
 c:\msdev\lib\msvcrt.lib    c:\msdev\lib\oldnames.lib
c:\msdev\lib\kernel32.lib     c:\msdev\lib\adv
api32.lib  c:\oracle\precomp\lib\msvc\orasql8.lib
c:\oracle\oci\lib\msvc\oci.lib
Microsoft (R) 32-Bit Incremental Linker Version 3.00.5270
Copyright (C) Microsoft Corp 1992-1995. All rights reserved.

   Creating library extproc.lib and object extproc.exp
```

Библиотека extproc.dll создана и готова для использования. Теперь давайте перенесем ее в среду ОС UNIX с помощью следующего файла управления проектом:

```
MAKEFILE= $(ORACLE_HOME)/rdbms/demo/demo_rdbms.m]c                    (1)

INCLUDE=  -I$(ORACLE_HOME)/rdbms/demo  \                              (2)
          -I$(ORACLE_HOME)/rdbms/public  \
          -I$(ORACLE_HOME)/plsql/public  \
          -I$(ORACLE_HOME)/network/public

TGTDLL= extproc.so                                                    (3)

OBJS  = extproc.o                                                     (4)

all: $(TGTDLL)                                                        (5)

clean:
    rm *.o                                                            (6)
$(TGTDLL): $(OBJS)
    $(MAKE) -f $(MAKEFILE) extproc_callback \                         (7)
     SHARED_LIBNAME=$(TGTDLL)OBJS=$(OBJS)

CC=cc                                                                 (8)
CFLAGS= -g -I. $(INCLUDE) -Wall                                       (9)
```

*И в этом случае выделенные **полужирным шрифтом** числа в круглых скобках не являются частью make-файла, а указаны лишь для возможности ссылок на них в дальнейшем.*

1. Имя/местонахождение стандартного **make**-файла Oracle. Я буду использовать этот файл для безошибочной компиляции и компоновки с необходимыми для данной платформы и версии Oracle библиотеками. Поскольку набор этих библиотек существенно отличается для каждого релиза, версии и платформы, я **настоятельно** рекомендую использовать этот make-файл.
2. Список каталогов для поиска включаемых файлов. Здесь я перечислил каталоги Oracle.
3. Имя создаваемой библиотеки.
4. Список файлов, образующих эту библиотеку.
5. Стандартная цель, которая будет создаваться.
6. Цель для удаления временных файлов, созданных в ходе сборки проекта.
7. Фактическая цель проекта. При ее построении для создания библиотеки внешних процедур используется стандартный make-файл, поставляемый корпорацией Oracle. Это снимает все проблемы с именами и местонахождением библиотек.
8. Имя компилятора языка C, который предполагается использовать.
9. Стандартный набор опций, которые необходимо передать компилятору языка C.

С учетом того, как был написан код, перенос закончен. Осталось выполнить команду **make** и получить примерно такой результат:

```
$ make
cc -g -I. -I/export/home/ora816/rdbms/demo -I/export/home/ora816/rdbms/
public -I/export/home/ora816/plsql/public -I/export/home/ora816/network/
public -Wall   -c extproc.c -o extproc.o
make  -f /export/home/ora816/rdbms/demo/demo_rdbms.mk extproc_callback \
 SHARED_LIBNAME=extproc.SO  OBJS=extproc.o
make[l]: Entering directory Varia-export/home/tkyte/src/demo_passing'
ld -G -L/export/home/ora816/lib -R/export/home/ora816/lib -o extproc.so
extproc.o -lclntsh `sed -e 's/-ljava//g' /export/home/ora816/lib/
ldflags` -lnsgr8 -lnzjs8 -ln8 -ln18 -lnro8 `sed -e 's/-ljava//g' /
export/home/ora816/lib/ldflags` -lnsgr8 -lnzjs8 -ln8 -ln18 -lclient8
-lvsn8 -lwtc8 -lcommon8 -lgeneric8 -lwtc8 -lmm -lnls8 -lcore8 -lnls8 -
lcore8 -lnls8 `sed -e 's/-ljava//g' /export/home/ora816/lib/ldflags`
-lnsgr8 -lnzjs8 -ln8 -ln18 -lnro8 `sed -e 's/-ljava//g' /export/home/
ora816/lib/ldflags` -lnsgr8 -lnzjs8 -ln8 -ln18 -lclient8 -lvsn8 -
lvtc8 -lcommon8 -lgeneric8 -ltrace8 -lnls8 -lcore8 -lnls8 -lcore8 -
lnls8 -lclient8 -lvsn8 -lwtc8 -lcommon8 -lgeneric8 -lnls8 -lcore8 -
lnls8 -lcore8 -lnls8 `cat /export/home/ora816/lib/sysliblist` `if [ -f
/usr/lib/libsched.so ] ; then echo -lsched ; else true; fi` -R/export/
home/ora816/lib -laio -lposix4 -lkstat -lm -lthread \
/export/home/ora816/lib/libpls8.a
makef[1]: Leaving directory '/aria-export/home/tkyte/src/demo_passing'
```

В результате, мы получили файл библиотеки **extproc.so** для ОС Solaris.

Установка и запуск

Теперь, при наличии спецификации вызова, объекта-библиотеки, соответствующих типов данных, спецификации и тела пакета demo_passing в файле extproc.sql, а также библиотеки extproc.dll (или extproc.so), все готово для установки нашего демонстрационного примера в базе данных. Для этого мы выполним команду @extproc.sql, а затем - ряд анонимных блоков, чтобы проверить работу внешних процедур. Необходимо настроить оператор CREATE LIBRARY так, чтобы он указывал на созданную .dll- или .so-библиотеку:

```
create or replace library demoPassing as
'C: \<местонахождение DLL-библиотеки>\extproc. d11';
```

но остальная часть должна компилироваться без изменений.

Итак, после запуска сценария extproc.sql можно проверить работу внешних процедур следующим образом:

```
SQL> declare
  2      l_input    number;
  3      l_output number;
  4  begin
  5       dbms_output.put_line('Передаем Number');
  6
  7    dbms_output.put_line('Сначала проверяем передачу пустых
-> значений');
  8      demo_passing_pkg.pass(l_input, l_output);
  9    dbms_output.put_line('l_input = '||l_input||
-> ' l_output = '||l_output);
 10
 11     l_input := 123;
 12    dbms_output.put_line ('Теперь проверяем передачу непустых
-> значений');
 13    dbms_output.put_line('Предполагается, что в результате будет -
-> 123');
 14      demo_passing_pkg.pass(l_input, l_output);
 15 dbms_output.put_line('l_input = ' || l_input || ' l_output =
->  '||l_output);
 16  end;
 17  /
Передаем Number
Сначала проверяем передачу пустых значений
l_input -  l_output -
Теперь проверяем передачу непустых значений
Предполагается, что в результате будет -123
l_input - 123 l_output - -123

PL/SQL procedure successfully completed.
```

Тексты сообщений в тестовом примере переведены на русский язык. Если вы будете использовать соответствующие коды с сайта издательства Wrox (http://www.wrox.com), сообщения будут выдаваться на английском языке. - Прим. научн. ред.

У меня есть простой анонимный блок для поочередной проверки каждой из созданных процедур и функций. Я не буду приводить здесь результат выполнения каждой команды. В код демонстрационного примера на сайте издательства включен сценарий **test_all.sql,** выполняющий каждую процедуру и функцию и выдающий результат, подобный представленному выше. После установки можно выполнить его, чтобы увидеть в работе каждую из внешних процедур.

Теперь, если вы помните код на языке C, там был ряд вызовов **debugf.** Если после выполнения представленного выше PL/SQL-блока просмотреть файл **ext_proc.log** во временном каталоге, можно увидеть результаты вызовов **debugf.** Они будут иметь следующий вид:

```
000809 185056 GMT (     extproc.c,176) Входим в функцию Pass Number
000809 185056 GMT (     extproc.c,183) Получена среда Oci
000809 185056 GMT (     extproc.c,176) Входим в функцию Pass Number
000809 185056 GMT (     extproc.c,183) Получена среда Oci
000809 185056 GMT (     extproc.c,209) Первый параметр: 123
000809 185056 GMT (     extproc.c,230) Устанавливаем параметр OUT равным -
123, а индикаторную переменную — равной NOTNULL
```

Это показывает, что 9 августа 2000 года (000809) в 6:50:56 вечера **(185056)** по Гринвичу (GMT) из строки 176 исходного кода в файле extproc.c был выполнен вызов **debugf** с сообщением "Входим в функцию Pass Number". Далее записаны все остальные выполненные вызовы **debugf.** Как видите, при отладке очень удобно использовать трассировочный файл, запись информации в который можно включать и отключать при необходимости. Поскольку внешние процедуры выполняются на **сервере,** отладка их может оказаться очень сложным делом. Хотя возможность использовать обычный отладчик не исключена, на практике к ней прибегают очень редко.

Внешняя процедура для сброса большого объекта в файл (LOB_IO)

В Oracle 8.0 появился ряд новых типов данных:

- **CLOB** — Character Large Object (символьный большой объект);
- **BLOB** — Binary Large Object (двоичный большой объект);
- **BFILE** - Binary FILE (двоичный файл).

Типы данных **CLOB** и **BLOB** позволяют сохранить в базе неструктурированные данные размером до 4 Гбайт. С помощью данных типа **BFILE** можно читать файлы ОС, находящиеся в файловой системе сервера. В составе сервера Oracle поставляется пакет **DBMS_LOB**, содержащий много утилит для работы с большими объектами. Он включает даже функцию loadfromfile для загрузки большого объекта из существующего файла ОС. Но в составе сервера Oracle, однако, не поставляется функция для записи большого объекта в файл ОС. Во многих случаях для данных типа **CLOB** можно использовать средства пакета **UTL_FILE**, но для данных типа **BLOB** это решение абсолютно не подходит. Сейчас мы займемся реализацией функции для записи в файл данных типа CLOB и BLOB в виде внешней процедуры на языке C, использующей средства прекомпилятора Pro*C.

Спецификация пакета LOB_IO

Снова начнем с оператора **CREATE LIBRARY**, затем определим спецификацию пакета, потом — тело пакета, задающее соответствующие подпрограммы для внешних С-функции, и, наконец, реализуем эти функции на языке С с помощью прекомпилятора Pro*C. Создаем библиотеку в базе данных:

```
tkyte@KYTE816> create or replace library lobToFile_lib
  2  as 'C:\extproc\lobtofile\extproc.dll'
  3  /
Library created.
```

Теперь переходим к спецификации создаваемого пакета. Она начинается с трех перегруженных функций для записи большого объекта в файл на сервере. Они вызываются одинаково и возвращают количество байтов, записанных на диск. После спецификаций перечислены исключительные ситуации, которые могут возбуждаться по ходу работы этих функций.

```
tkyte@TKYTE816> create or replace package lob_io
  2  as
  3
  4      function write(p_path in varchar2,
  5                     p_filename in varchar2, p_lob in blob)
  6      return binary_integer;
  7
  8      function write(p_path in varchar2,
  9                     p_filename in varchar2, p_lob in clob)
 10      return binary_integer;
 11
 12      function write(p_path in varchar2,
 13                     p_filename in varchar2, p_lob in bfile)
 14      return binary_integer;
 15
 16      IO_ERROR exception;
 17     pragma exception_init(IO_ERROR, -20001);
 18
 19      CONNECT_ERROR exception;
 20     pragma exception_init(CONNECT_ERROR, -20002);
 21
 22      INVALID_LOB exception;
 23     pragma exception_init(INVALID_LOB, -20003);
 24
 25      INVALID_FILENAME exception;
 26     pragma exception_init(INVALID_FILENAME, -20004);
 27
 28      OPEN_FILE_ERROR exception;
 29     pragma exception_init(OPEN_FILE_ERROR, -20005);
 30
 31      LOB_READ_ERROR exception;
 32     pragma exception_init(LOB_READ_ERROR, -20006);
 33
```

```
 34  end;
 35  /
Package created.
```

Здесь мы каждому коду ошибки (эти коды ошибки определены с помощью директив **#define ERROR_** в начале файла с исходным текстом внешних процедур) сопоставляем имя исключительной ситуации PL/SQL. Это удобное дополнение, позволяющее пользователю пакета перехватывать исключительные ситуации с определенными **именами** следующим образом:

```
exception
    when lob_io.IO_ERR0R then
            •••
    when lob_io.CONNECT_ERR0R then
            •••
```

или при желании получать вместо исключительных ситуаций коды ошибок и тексты сообщений об ошибках:

```
exception
    when others then
        if (sqlcode = -20001 ) then — (это была ошибка ввода/вывода)
            ...
       elsif( sqlcode = -20002 )  then — (это была ошибка подключения)
            ... и так далее
```

Даже не просматривая соответствующий код на языке C, легко понять, какие ошибки могут возникнуть во внешней процедуре

Теперь переходим к телу пакета. В нем каждой из представленных выше спецификаций PL/SQL-функций сопоставляется C-функция из библиотеки lobToFile:

```
tkyte@TKYTE816> create or replace package body lob_io
  2  as
  3
  4  function write(p_path in varchar2, p_filename in varchar2, p_lob in
-> blob)
  5  return binary_integer
  6  as
  7  language C name "lobToFile" library lobtofile_lib
  8  with context parameters (CONTEXT,
  9     p_path     STRING,         p_path     INDICATOR short,
 10    p_filename STRING,         p_filename INDICATOR short,
 11    p_lob      OCILOBLOCATOR, p_lob      INDICATOR short,
 12    RETURN INDICATOR short);
 13
 14
 15  function write(p_path in varchar2, p_filename in varchar2, p_lob in
-> clob)
 16  return binary_integer
 17  as
 18  language C name "lobToFile" library lobtofile_lib
 19  with context parameters (CONTEXT,
 20    p_path     STRING,         p_path     INDICATOR short,
```

```
 21    p_filename STRING,        p_filename INDICATOR short,
 22    p_lob      OCILOBLOCATOR, p_lob      INDICATOR short,
 23    RETURN INDICATOR short);
 24
 25
 26 function write(p_path in varchar2, p_filename in varchar2, p_lob in-
-> bfile)
 27 return binary_integer
 28 as
 29 language C name "lobToFile" library lobtofile_lib
 30 with context parameters (CONTEXT,
 31    p_path     STRING,        p_path     INDICATOR short,
 32    p_filename STRING,    p_filename     INDICATOR short,
 33    p_lob      OCILOBLOCATOR, p_lob      INDICATOR short,
 34    RETURN INDICATOR short);
 35
 36 end lob_io;
 37 /

Package body created.
```

Интересно отметить, что все три функции сопоставляются **одной и той же внешней C-функции.** Я не писал отдельные функции для данных типа **CLOB, BLOB** и **BFILE.** Поскольку любой большой объект передается как данные типа **OCILOBLOCATOR,** все их можно обрабатывать одной и той же функцией. Как обычно, я передаю индикаторную переменную для каждого формального параметра и для возвращаемого значения. Хотя это и не обязательно, но очень рекомендуется.

Код Pro*C для пакета LOB_IO

Теперь рассмотрим код для прекомпилятора Pro*C, реализующий библиотеку **Iobtoflle_lib. Я** не буду комментировать универсальный код, рассмотренный в первом примере, чтобы сократить текст (функции **debugf, raise_application_error, ociLastError, term** и **init** — такие же, за исключением того, что в функции **init** в приложениях на Pro'C используется конструкция **EXEC SQL REGISTER CONNECT),** и перейду сразу к коду специфическому. Следует отметить, что представленный далее код должен идти после представленного ранее "шаблонного" кода, и что в шаблоне необходимо убрать комментарии с разделов, связанных с подключением в Pro*C. Начнем с определения всех ошибок, о которых будет выдаваться сообщение. Этот набор кодов ошибок должен в точности соответствовать кодам ошибок для исключительных ситуаций, заданных в спецификации PL/SQL-пакета. Гарантировать это соответствие невозможно; это всего лишь договоренность, которой я привык следовать, но следовать ей, определенно, стоит.

```
#define ERROR_FWRITE            20001
#define ERROR_REGISTER_CONNECT  20002
#define ERROR_BLOB_IS_NULL      20003
#define ERROR_FILENAME_IS_NULL  20004
#define ERROR_OPEN_FILE         20005
#define ERROR_LOB_READ          20006
```

Дальше идет внутренняя функция, непосредственно из PL/SQL недоступная, которая будет использоваться основной функцией lobToFile для записи данных в файл. Она также подсчитывает количество байтов, записанных в файл:

```
static int writeToFile(myCtxStruct *       myCtx,
                       OCIFileObject *     output,
                       char *              buff,
                       int                 bytes,
                       int *               totalWritten)
{
ub4   bytesWritten;

    debugf(myCtx, "Записываем %d байтов в файл", bytes);
    if(OCIFileWrite(myCtx->envhp,myCtx->errhp,output,
                    buff, bytes, &bytesWritten) != OCI_SUCCESS)
    {
        return raise_application_error
               (myCtx,
                ERROR_FWRITE,
                "Error writing to file '%s'",
                lastOciError(myCtx));
    }
    if (bytesWritten != bytes)
    {
        return raise_application_error
               (myCtx,
                ERROR_FWRITE,
                "Ошибка записи в файл %d байт, записано только %d байт",
                 bytes, bytesWritten);
    }
    *totalWritten+=bytesWritten;
    return 0;
}
```

Первый параметр этой функции — контекст сеанса. Этот контекст должен передаваться всем вызываемым функциям, чтобы можно было использовать такие утилиты, как raise_application_error. Следующий параметр — выходной файл, в который будут записываться данные. Для выполнения ввода/вывода используются переносимые функции OCIFile. Предполагается, что перед вызовом функции writeToFile соответствующий файл уже открыт. Далее идут указатели на записываемый буфер и количество байтов в буфере. Последней передается переменная-счетчик, в которой хранится общее количество записанных байтов.

Теперь переходим к основной (и последней) функции. Эта функция выполняет все необходимые действия; она принимает локатор большого объекта (независимо от типа объекта — BLOB, CLOB или BFILE) и записывает его содержимое в указанный файл:

```
#ifdef WIN_NT
_declspec  (dllexport)
#endif
int lobToFile(OCIExtProcContext * ctx,
              char *              path,
```

```
                short               path_i,
                char *              filename,
                short               filename_i,
                OCIBlobLocator  *   blob,
                short               blob_i,
                short *             return_indicator)
{
```

Следующая часть кода задает структуру, в которую мы будем выбирать данные. Она содержит начальное поле размера, в байтах, а затем — пространство для данных размером 64 Кбайт. Мы будем выбирать данные из большого объекта порциями по 64 Кбайт и записывать их на диск. Затем определяются необходимые локальные переменные:

```
typedef struct long_varraw
{
  ub4  len;
  text buf[65536];
} long_varraw;
EXEC SQL TYPE long_varraw IS LONG VARRAW(65536);

long_varraw    data;     /* в эту структуру мы будем выбирать данные */
ub4            amt;     /* здесь будет храниться количество выбранных */
                        /* байтов */
ub4           buffsize = sizeof(data.buf);  /* это количество байтов */
                        /* мы запрашиваем */
int            offset = 1; /* с какой позиции большого объекта мы */
                        /* читаем данные */
OCIFileObject* output = NULL; /* файл, в который выполняется запись */
int           bytesWritten = 0; /* сколько байтов всего ЗАПИСАНО */
myCtxStruct *  myCtx;

    *return_indicator = OCI_IND_NULL;
    if ((myCtx=init(ctx)) = NULL) return 0;
```

Начнем с проверки индикаторов пустых значений. Если установлена любая из индикаторных переменных, необходимо вернуть сообщение об ошибке. Это показывает, почему важно **всегда** передавать индикаторные переменные во внешние процедуры на языке C. Никогда нельзя быть уверенным, что пользователь случайно не передал пустое значение. Попытка без предварительной проверки обратиться к файлу с указанным именем или к большому объекту, которые окажутся пустыми, может закончиться катастрофически (внешняя процедура закончится неудачно), поскольку параметры окажутся не проинициализированными.

```
    if (blob_i == OCI_IND_NULL)
    {
        raise_application_error
               (myCtx,
                ERRQR_BLOB_IS_NULL,
                "Функции lobToFile передан пустой большой объект;
->недопустимый аргумент");
    )
    else if (filename_i == OCI_IND_NULL || path_i = OCI_IND_NULL)
```

```
    {
        raise_application_error
               (myCtx,
                ERROR_FILENAME_IS_NULL,
               "Функции lobToFile передано пустое имя файла или каталога;
->недопустимыйаргумент");
    }
```

Теперь откроем файл. Мы открываем его на запись в двоичном режиме. Мы хотим просто сбросить байты из базы данных в файл.

```
    else if (OCIFileOpen(myCtx->envhp, myCtx->errhp, &output,
                         filename,path,
                         OCI_FILE_WRITE_ONLY, OCI_FILE_CREATE,
                         OCI_FILE_BIN) != OCI_SUCCESS)
    {
        raise_application_error(myCtx,
                                ERROR_OPEN_FILE,
                               "Ошибка открытия файла '%s'",
                                lastOciError(myCtx));
    }
    else
    {
        debugf(myCtx, "lobToFile(filename => '%s%s', lob => %X)",
               path, filename, blob);
```

Теперь мы будем читать большой объект с помощью средств Pro*C методом **без** опроса (non-polling). Это важно, поскольку "опрашивать" большой объект во внешней процедуре нельзя. Таким образом, мы никогда не запросим больше данных, чем можем получить в одном вызове (non-polling). Мы начинаем со смещения 1 (с первого байта) и будем читать по **BUFSIZE** байтов за раз (64 Кбайт в данном случае). Каждый раз увеличивая смещение на прочитанное количество байтов, мы выйдем из цикла только когда считано будет столько байтов, сколько запрошено — это будет означать, что прочитан весь большой объект.

```
        for( offset = 1, amt = buffsize;
             amt = buffsize;
            offset += amt )
        {
            debugf(myCtx,
                "Попытка прочитать %d байт из большого объекта ", amt);
            EXEC SQL LOB
                      READ :amt
                      FROM :blob
                      AT   :offset
                      INTO :data
                      WITH LENGTH :buffsize;
```

Проверяйте **все** возможные ошибки — при их возникновении выдавайте **собственные** сообщения об ошибках в стек ошибок среды PL/SQL.

Обратите внимание, как мы освобождаем все использованные ресурсы (открытый файл) перед завершением работы. Это важно. По возможности, надо предотвращать

утечку ресурсов. Для этого мы возвращаем управление только в одном месте (ниже) и перед этим вызываем функцию term:

```
            if (sqlca.sqlcode < 0)
                break;

            if (writeToFile(myCtx, output, data.buf, amt, &bytesWritte
                break;
        >
    }
```

Осталось закрыть файл и вернуть управление:

```
    if (output != NULL)
    {
        debugf(myCtx, "Закончили запись и закрываем файл");
        OCIFileClose(myCtx->envhp, myCtx->errhp, output);
    }
    *return_indicator = OCI_IND_NOTNULL;
    debugf(myCtx,
           "Возвращаем значение %d как количество прочитанных байтов",
           bytesWritten);

    term(myCtx);
    return bytesWritten;
}
```

Создание внешней процедуры

Процесс создания библиотеки **lobtofile** почти совпадает с рассмотренным ранее для библиотеки **demo_passing.** Универсальный файл управления проектом (make-файл) использовался как в среде Windows, так и в ОС UNIX с минимальными изменениями. В Windows мы используем:

```
CPU=i386

MSDEV       = c:\msdev
ORACLE_H0ME = c:\oracle

!include <$(MSDEV)\include\win32.mak>

TGTDLL = extproc.dll
OBJS   = lobtofile.obj

NTUSER32LIBS    = $(MSDEV)\lib\user32.lib   \
                  $ (MSDEV)\lib\msvcrt.lib  \
                  $(MSDEV)\lib\oldnames.lib  \
                  $(MSDEV)\lib\kernel32.lib   \
                  $(MSDEV)\lib\advapi32.lib

SQLLIB  =  $(0RACLE_H0ME)\precomp\lib\msvc\orasql8.1ib  \
           $(ORACLE_HOME)\oci\lib\msvc\oci.lib

INCLS   = -I$(MSDEV)\include \
          -I$(ORACLE_HOME)\oci\include \
          -I.
```

```
CFIAGS  = $(INCLS)  -DWIN32 -DWIN_NT -D_DLL

all: $(TGTDLL)

clean:
    erase *.obj *.lib *.exp lobtofile.c

$(TGTDLL): $(OBJS)
    $(link) -DLL $(dllflags) \
        /NODEFAULTLIB:LIBC.LIB -out:$(TGTDLL)  \
        $(OBJS) \
        $(NTUSER32LIBS) \
        ${SQLLIB) \

lobtofile.c: lobtofile.pc
       proc \
        lnclude=$(ORACLE_HOME)\network\public  \
        include=$(ORACXE_HCME)\proc\lib  \
        include=$(ORACLE_HCME)\rdbnis\deroo  \
        lnclude=$(ORACLE_HCME)\oci\lnclude  \
        include=$(MSDEV) \include \
        lines=yes \
       parse=full \
        iname=lobtofile.pc
```

Изменения выделены **полужирным** шрифтом. Было изменено имя компонуемого объектного файла и добавлено правило для автоматического преобразования **lobtofile.pc** в lobtofile.c. Вызванному прекомпилятору Pro*C мы сообщаем, где находятся заголовочные файлы **(INCLUDE=),** что номера строк следует сохранить в полученном .c-файле (lines=yes), что требуется проанализировать код на языке C **(parse=full)** и что имя преобразуемого файла — **lobtofile.pc (iname=).** Теперь осталось выполнить команду **nmake,** и DLL-библиотека будет создана.

В ОС UNIX make-файл имеет следующий вид:

```
MAKEFILE=  $(ORACLE_HOME)/rdbms/demo/demo_rdbms.mk

INCLUDE= -I$(ORACLE_HOME)/rdbms/demo \
         -I$(ORACLE_HOME)/rdbms/public \
         -I$(ORACLE_HOME)/plsql/public \
         -I$(ORACLE_HOME)/network/public

TGTDLL= extproc.so
OBJS  = lobtofile.o

all: $(TGTDLL)

clean:
    rm *.o

lobtofile.c: lobtofile.pc
    proc  \
         include=$(ORACLE_HOME)/network/public  \
         include=$(ORACLE_HOME)/proc/lib  \
         include=$(ORACLE_HOME)/rdbms/deino  \
         include=$(ORACLE_HOME)/rdbms/public  \
```

```
        lines=yes \
        lname=lobtofile.pc

extproc.so: lobtofile.c lobtofile.o
    $(MAKE) -f $(MAKEFILE) extproc_callback \
     SHARED_LIBNAME=extproc.so  OBJS="lobtofile.o"

CC=cc
CFIAGS= -g -I. $(INCLUDE)
```

Для ОС UNIX мы сделали такие же изменения, как и в среде Windows. Мы просто добавили команду для вызова прекомпилятора Pro*C и изменили имя компонуемого объектного файла. Набираем команду make и получаем соответствующий .so-файл.

Теперь все готово для его проверки и использования.

Установка и использование пакета LOB_IO

Осталось только выполнить операторы **CREATE LIBRARY**, **CREATE PACKAGE** и **CREATE PACKAGE BODY**. При этом пакет **LOB_IO** будет установлен в базе данных. Для его тестирования мы используем пару анонимных PL/SQL-блоков. Первый блок будет проверять средства выявления и обработки ошибок. Вызовем внешнюю процедуру и намеренно передадим ей некорректные входные данные, неверные имена каталогов и т.п. Вот этот блок с комментариями, описывающими, что мы должны получить на каждом шаге:

```
SQL> REM для NT
SQL> REM задайте PATH=c:\tesap\
SQL> REM задайте CMD=fc /b

SQL> REM для UNIX
SQL> задайте PATH=/tmp/
SQL> задайте CMD="diff -s"

SQL> drop table demo;
Table dropped.

SQL> create table demo(theBlob blob, theClob clob);
Table created.

SQL> /*
DOC>* В следующем блоке проверятся вое условия возникновения выявленных
DOC> * нами ошибок. Не проверяется ошибки IO_ERROR (для этого надо, чтобы
DOC> * на диске не осталось свободного места или запись была невозможна
DOC> * по другой подобной причине) и CONNECT_ERROR (это не должно
DOC> * случиться *никогда*)
DOC>*/
SQL>
SQL> declare
  2      l_blob    blob;
  3      l_bytes number;
  4  begin
  5
  6      /*
  7       * Пытаемся передать большой объект NULL
```

```
 8        •/
 9       begin
10           l_bytes := lob_io.write('&PATH', 'test.dat', l_blob);
11       exception
12           when lob_io.INVALID_LOB then
13                dbms_output.put_line('недопустимей аргумент перехвачен,
-> как и ожидалось');
14                dbms_output.put_line(rpad('-',70,'-'));
15       end;
16
17       /•
18        * Теперь попытаемся передать реальный большой объект и имя
-> файла MOLL
19        */
20       begin
21           insert into demo (theBlob) values(empty_blob())
22           returning theBlob into l_blob;
23
24          l_bytes := lob_io.write(NULL, NULL, l_blob);
25       exception
26           when lob_io.INVALID_FILENAME then
27                dbms_output.put_line('недопустимый аргумент снова
перехвачен, как и ожидалось');
28               dbms_output.put_line(rpad('-',70,'-'));
29      end;
30
31       /*
32        * Теперь попытаемся передать существующий большой объект, во
-> несуществующий каталог
33        */
34       begin
35        l_bytes      := lob_io.write('/nonexistent/directory', 'x.dat',
-> l_blob);
36       exception
37           when lob_io.OPEN_FILE_ERROR then
38                dbms_output.put_line('перехватили ошибку открытия файла,
-> как и ожидалось');
39               dbms_output.put_line(sqlerrm);
40               dbms_output.put_line(rpad('-',70,'-'));
41      end;
42
43       /*
44        * Теперь запишем объект, чтобы проверить работу функции
45        */
46       l_bytes := lob_io.write('&PATH', 'l.dat', l_blob);
47        dbms_output.put_line('Успешно записали ' || l_bytes ||
-> 'байтов');
48        dbms_output.put_line(rpad('-',70, '-'));
49
50       rollback;
51
```

```
 52     /*
 53      * Теперь у нас есть непустой большой объект, НО мы ВЫПОЛНИЛИ
 54  *  откат, так что локатор большого объекта стал недействительны.
 55       * Давайте посмотрим, что выдаст внешняя процедура в данной
-> случае...
 56       V
 57      begin
 58          l_bytes  := lob_io.write('&PATH', '1.dat', l_blob);
 59      exception
 60          when lob_io.LOB_READ_ERROR then
 61              dbms_output.put_line('перехватили ошибку чтения большого
-> объекта, как и ожидалось');
 62              dbms_output.put_line(sqlerrm) ;
 63              dbms_output.put_line(rpad('-',70,'-'));
 64     end;
 65 end;
 66 /
old  10:         l_bytes  := lob_io.write(&'PATH', 'test.dat', l_blob);
new  10:        l_bytes  := lob_io.write('/tmp/', 'test.dat', l_blob);
old  46:     l_bytes  := lob_io.write('&PATH', '1.dat', l_blob);
new  46:    l_bytes  := lob_io.write('/tmp/', '1.dat', l_blob);
old  58:        l_bytes  := lob_io.write('&PATH', '1.dat', l_blob);
new  58:         l_bytes  := lob_io.write('/tmp/', '1.dat', l_blob);

недопустимый аргумент перехвачен, как и ожидалось

недопустимый аргумент снова перехвачен, как и ожидалось

перехватили ошибку открытия файла, как и ожидалось
ORA-20005: Error opening file 'ORA-30152: File does not exist'

Успешно записали 0 байт


PL/SQL procedure successfully completed.
```

Как видите, все произошло именно так, как и ожидалось. Мы намеренно сделали несколько ошибок и получили соответствующие сообщения. Теперь используем пакет по прямому назначению. Для этого теста я создал объект-каталог в базе данных, соответствующий моему временному каталогу (**/tmp** в ОС UNIX, **C:\temp** в Windows). Объект-каталог используется при работе с данными типа **BFILE,** позволяя читать файлы в указанном каталоге. В файловую систему ОС (**/tmp** или **C:\temp**) я помещу тестовый файл **something.big.** Это достаточно большой файл для проверки внешней процедуры. Его содержимое не имеет значения. Мы загрузим этот файл в столбец типа **CLOB,** затем — в столбец типа **BLOB** и, наконец, во временный большой объект. Затем запишем каждый из этих больших объектов в отдельный файл с помощью созданной внешней процедуры. В завершение с помощью утилит ОС (**diff** в UNIX и FC в Windows) сравним сгенерированные файлы с исходным:

```
SQL> create or replace directory my_files as '&PATH.';
old   1: create or replace directory my_files as '&PATH.'
new   1: create or replace directory my_files as '/tmp/'
```

```
Directory created.
SQL>
SQL> declare
  2     l_blob     blob;
  3      l_clob    clob;
  4      l_bfile    bfile;
  5  begin
  6      insert into demo
  7     values (empty_blob(), empty_clob())
  8     returning theBlob, theClob into l_blob, l_clob;
  9
 10     l_bfile := bfilename ('MY_FILES', 'something.big');
 11
 12    dbms_lob.fileopen(l_bfile);
 13
 14     dbms_lob.loadfromfile(l_blob, l_bfile,
 15                           dbms_lob.getlength(l_bfile));
 16
 17     dbms_lob.loadfromfile(l_clob, l_bfile,
 18                           dbms_lob.getlength(l_bfile));
 19
 20    dbms_lob.fileclose(l_bfile);
 21      commit;
 22  end;
 23  /

PL/SQL procedure successfully completed.
```

Итак, мы загрузили файл something.big в базу данных (сначала — как данные типа BLOB, затем — как данные типа CLOB). Теперь снова запишем содержимое этих больших объектов во внешние файлы:

```
SQL> declare
  2      l_bytes number;
  3      l_bfile    bfile;
  4  begin
  5      for x in (select theBlob from demo)
  6      loop
 .7         l_bytes := lob_io.write('&PATH','blob.dat', x.theBlob);
  8         dbms_output.put_line('Записали ' || l_bytes || ' байтов
-> данных типа blob');
  9      end loop;
 10
 11      for x in (select theClob from demo)
 12      loop
 13          l_bytes := lob_io.write('&PATH','clob.dat', x.theclob);
 14          dbms_output.put_line('Записали ' || l_bytes ||' байтов
-> данных типа clob');
 15       end loop;
 16
 17      l_bfile := bfilename('MY_FILES', 'something.big');
 18      dbms_lob.fileopen(l_bfile);
 19      1 bytes := lob io.write('&PATH','bfile.dat', l_bfile);
```

```
 20      dbms_output.put_line('Записали ' || l_bytes || ' байтов данных
-> типа bfile');
 21      dbms_lob.fileclose(l_bfile);
 22  end;
 23  /
old   7:         l_bytes := lob_io.write('&PATH','blob.dat', x.theBlob);
new   7:         l_bytes := lob_io.write('/tmp/', 'blob.dat', x.theBlob);
old  13:         l_bytes :=lob_io.write('&PATH','clob.dat',x.theClob);
new  13:         l_bytes := lob_io.write('/tmp/', 'clob.dat', x.theClob);
old  19:     l_bytes := lob_io.write('SPATH','bfile.dat', l_bfile);
new  19:    l_bytes   := lob_io.write('/tmp/','bfile.dat', l_bfile);
Записали 1107317 байт данных типа blob
Записали 1107317 байт данных типа clob
Записали 1107317 байт данных типа bfile

PL/SQL procedure successfully completed.
```

Это показывает, что мы успешно вызвали внешнюю процедуру и трижды записали файл. Каждый раз размер файла был одним и тем же (как и ожидалось). Теперь создадим временный большой объект, скопируем в него файл и запишем объект в другой файл, чтобы убедиться, что можно работать и с временными большими объектами:

```
SQL> declare
  2      l_tmpblob blob;
  3      l_blob    blob;
  4      l_bytes      number;
  5  begin
  6    select theBlob into l_blob from demo;
  7
  8    dbms_lob.createtemporary(l_tmpblob,TRUE);
  9
 10   dbms_lob.copy(l_tmpblob,l_blob,dbms_lob.getlength(l_blob),1,1);
 11
 12   l_bytes   := lob_io.write('&PATH','tempblob.dat', l_tmpblob);
 13    dbms_output.put_line('Записали ' || l_bytes || ' байтов временного
-> большого объекта');
 14
 15      DBMS_LOB.FREETEMPORARY(l_tmpblob);
 16  END;
 17  /
old  12:     l_bytes := lob_io.write('&PATH','tempblob.dat', l_tmpblob);
new  12:     l_bytes := lob_io.write('/tmp/','tempblob.dat', l_tmpblob);
Записали 1107317 байтов временного большого объекта

PL/SQL procedure successfully completed.
```

Таким образом, запись прошла успешно и, к счастью, записано точно такое же количество байтов. Последний шаг — проверить с помощью утилит ОС, что записанные файлы идентичны загруженному файлу:

```
SQL> host &CMD SPATH.something.big &PATH.blob.dat
Files /tmp/something.big and /tmp/blob.dat are identical

SQL> host &CMD &PATH.something.big &PATH.clob.dat
Files /tmp/something.big and /tmp/clob.dat are identical
```

```
SQL> host &CMD &PATH.something.big &PATH.bfile.dat
Files /tmp/something.big and /tmp/bfile.dat are identical

SQL> host &CMD &PATH.something.big &PATH.tempblob.dat
Files /tmp/something.big and /tmp/tempblob.dat are identical
```

Мы проверили работу нового пакета LOB_IO.

Возможные ошибки

Ниже представлен список типичных ошибок, которые могут возникнуть при использовании внешних процедур. Некоторые из них мы уже обсуждали, например ошибку, возникающую при неправильном конфигурировании процесса прослушивания или файла TNSNAMES.ORA. Но многие ошибки еше не рассмотрены. Мы займемся ими сейчас: я расскажу, когда они могут возникать и что сделать, чтобы их исправить.

Все эти сообщения об ошибках описаны также в руководстве *Oracle 8i Error Messages Manual.

ORA-28575 "невозможно открыть соединение RPC с агентом внешней процедуры"

```
28575, 00000, "unable to open RPC connection to external procedure agent"
// * Причина:  инициализация подключения по сети к агенту внешних
//             процедур не прошла успешно. Это может быть связано с
//             проблемами сети, неправильной конфигурацией процесса
//             прослушивания или некорректным кодом переноса.
// * Действие: проверить конфигурацию процесса прослушивания в файлах
//             LISTENER.ORA и TNSNAMES.ORA или проверить сервер Oracle
//             Names.
```

Эта ошибка почти всегда свидетельствует об ошибках конфигурации в файлах TNSNAMES.ORA или LISTENER.ORA. Возможные причины ее возникновения уже рассматривались ранее, в разделе "Конфигурирование сервера".

ORA-28576 "потеряно соединение RPC с агентом внешней процедуры"

```
28576, 00000, "lost RPC connection to external procedure agent"
// * Причина:  произошла фатальная ошибка сетевого соединения, в агенте
//             внешних процедур или в вызванной внешней процедуре после
//             успешной организации взаимодействия.
// * Действие: сначала проверьте вызываемый код внешней процедуры,
//             поскольку наиболее вероятной причиной получения этого
//             сообщения об ошибке является аварийное завершение
//             вызванной С-функции. Если — нет, проверьте сеть. Устраните
//             проблему, если она обнаружена. Если все компоненты
//             функционируют нормально, но проблема остается, она может
//             быть связана с внутренней логической ошибкой в коде
//             передачи RPC (вызова удаленной процедуры). Свяжитесь с
//             представителем службы поддержки.
```

** Текст сообщения об ошибке сначала приведен так, как он выдается СУБД Oracle версии 8.0.5.0.0 при установке русского языка для сообщений. В примерах и описаниях оставлены сообщения на английском языке. - Прим. научн. ред.*

Это сообщение об ошибке при обращении к внешней процедуре почти наверняка связано с ошибкой в разработанном вами коде. Эта ошибка возникает, когда "исчезает" внешний процесс. Это происходит в случае "слета" программы. Например, я добавил строку:

```
char * return_string
  (OCIExtProcContext * ctx,
   short *             return_i,
   int   *             return_l)
{

    *return_i = OCI_IND_NOTNULL;
    *(char*)NDLL = 1;
    return return_value;
}
```

в конце текста функции return_string. После перекомпиляции обнаружилось следующее:

```
ops$tkyte@ORA816.US.ORACLE.COM> exec dbms_output.put_line(
demo_passing_pkg.return_string)
BEGIN dbms_output.put_line(demo_passing_pkg.return_string); END;
*
ERROR at line 1:
ORA-28576: lost RPC connection to external procedure agent
```

Это сообщение об ошибке будет выдаваться до тех пор, пока вы не устраните ошибку в исходном коде.

ORA-28577 "аргумент %s внешней процедуры %s имеет неподдерживаемый тип данных %s"

```
28577, 00000, "argument %s of external procedure %s has unsupported
->datatype %s"
// * Причина:  при передаче аргументов внешней процедуры агенту обнаружен
//             тип данных, не поддерживаемый системой.
// * Действие: найдите в документации список поддерживаемых типов данных
//             аргументов внешних процедур.
```

Эта ошибка возникает при попытке передать из PL/SQL во внешнюю процедуру тип данных, не поддерживаемый данным интерфейсом. В частности, это относится к PL/SQL-таблицам. Если в примере demo_passing объявить тип numArray в спецификации пакета:

```
type numArray is table of number index by binary_integer;
procedure pass(p_in in numArray, p_out out numArray);
```

а не как тип вложенной таблицы SQL, во время выполнения произойдет следующее:

```
1  declare
2      l_input    demo_passing_pkg.numArray;
3      l_output   demo_passing_pkg.numArray;
4  begin
5      demo_passing_pkg.pass(l_input, l_output);
```

```
  6* end;
SQL> /
declare

ERROR at line 1:
ORA-28577: argument 2 of external procedure pass_numArray has unsupported
datatype
ORA-06512: at "OPS$TKYTE.DEM0_PASSING_PKG", line 0
ORA-06512: at line 5
```

Причина в том, что передача PL/SQL-таблиц не поддерживается (можно передавать наборы, тип которых создан в базе данных, но не PL/SQL-таблицы).

ORA-28578 "ошибка протокола во время вызова внешней процедуры"

```
28578, 00000, "protocol error during callback from an external procedure"
// * Причина:  произошла внутренняя ошибка протокола при попытке
//             выполнить обращение (callback) к серверу Oracle из
//             созданной пользователем внешней процедуры.
// * Действие: свяжитесь со службой поддержки Oracle.
```

К счастью, я никогда не получал ни этого, ни приведенного ниже сообщения об ошибке. Оно свидетельствует о внутренней ошибке сервера Oracle. Единственное, что можно сделать, получив это сообщение об ошибке, — попытаться воспроизвести его с помощью небольшого тестового кода и сообщить о проблеме службе поддержки Oracle.

ORA-28579 "сетевая ошибка во время вызова от агента внешней процедуры"

```
ORA-28579 "network error during callback from external procedure agent"
// * Причина:  произошла внутренняя ошибка сети при попытке выполнить
//             обращение (callback) к серверу Oracle из созданной
//             пользователем внешней процедуры.
// * Действие: свяжитесь со службой поддержки Oracle.
```

ORA-28580 "рекурсивные внешние процедуры не поддерживаются"

```
ORA-28580 "recursive external procedures are not supported"
// * Причина:  обращение из внешней процедуры привело к вызову другой
//             внешней процедуры.
// * Действие: проверьте, не вызывает ли выполняемый при обращении к
//             серверу SQL-код (непосредственно или косвенно, например
//             при срабатывании триггера) другую внешнюю процедуру, PL/
//             SQL-процедуру, вызывающую внешние процедуры и т.д.
```

Эта ошибка возникает при обращении из внешней процедуры к серверу, когда вызванная процедура обращается к другой внешней процедуре. Другими словами, внешняя процедура не может прямо или косвенно вызывать другую внешнюю процедуру. Это можно продемонстрировать, изменив .pc-файл для нашего пакета LOB_IO. Я добавил в этот файл следующее:

```
{int x;
exec sql execute begin
```

```
        :x := demo_passing_pkg.return_number;
    end; end-exec;

    if (sqlca.sqlcode < 0)
    {
        return raise_application_error
               (ctx,
                20000,
               "Error:\n%.70s",
                sqlca.sqlerrm.sqlerrmc);
    }
    }
```

сразу после вызова REGISTER CONNECT. Теперь при попытке выполнения основной функции пакета lob_io мы получим:

```
ops$tkyte@DEV8I.WORLD> declare x clob; y number; begin y :=
lob_io.write('x', x); end;
  2  /
declare x clob; y number; begin y := lob_io.write('x', x); end;
*
ERROR at line 1:
ORA-20000: Error:
ORA-28580: recursive external procedures are not supported
ORA-06512:
ORA-06512: at "OPS$TKYTE.LOB_IO", line 0
ORA-06512: at line 1
```

Единственное решение — никогда не вызывать из внешней процедуры другую внешнюю процедуру.

ORA-28582 "прямое соединение с этим агентом не разрешено"

```
$ oerr ora 28582
28582, 00000, "a direct connection to this agent is not allowed"
// * Причина:  пользователь или инструментальное средство пытается
//             непосредственно подключиться к агенту внешних процедур или
//             к агенту гетерогенных служб (Heterogeneous Services),
//             например:
//             "SVRMGR> CONNECT SCOTT/TIGER@NETWORK_ALIAS".
//             Такие подключения не разрешены.
//* Действие:  при выполнении оператора CONNECT проверьте, не ссылается
//             ли связь базы данных или псевдоним на агента гетерогенных
//             служб или агента внешних процедур.
```

Это сообщение может быть выдано только после попытки подключиться к базе данных, если случайно указано имя службы, сконфигурированной для подключения к службе внешних процедур.

ORA-06520 "PL/SQL: ошибка загрузки внешней библиотеки"

```
$ oerr ora 6520
06520, 00000, "PL/SQL: Error loading external library"
// * Причина:  выявлена ошибка, связанная с попыткой динамической
//             загрузки внешней библиотеки в PL/SQL.
```

```
// * Действие: другие сообщения об ошибках (если они есть) подскажут
//             причину выдачи этого сообщения.
```

После этого сообщения об ошибке должно быть выдано специфическое сообщение об ошибке ОС. Чтобы продемонстрировать эту ошибку, я сделал следующее:

```
$ cp lobtofile.pc extproc.so
```

То есть скопировал исходный код поверх .so-файла, что, определенно, должно вызвать проблемы. Теперь при попытке вызвать внешнюю процедуру я получаю:

```
declare x clob; y number; begin y := lob_io.write('x', x); end;

ERROR at line 1:
ORA-06520: PL/SQL: Error loading external library
ORA-06522: ld.so.1: extprocPLSExtProc: fatal: /export/home/tkyte/src/
lobtofile/extproc.so: unknown file type
ORA-06512: at "OPS$TKYTE.LOB_IO", line 0
ORA-06512: at line 1
```

Итак, как видите, среди сообщений об ошибках есть сообщение ОС, свидетельствующее о неизвестном типе файла, что и поможет выявить причину возникновения ошибки (в данном случае все просто: при просмотре файла extproc.se оказывается, что он содержит исходный код на языке C).

ORA-06521 "PL/SQL: ошибка отображения функции"

```
$ oerr ora 6521
06521, 00000, "PL/SQL: Error mapping function"
// * Причина:  ошибка возникла при попытке динамического сопоставления с
//             указанной функцией в PL/SQL.
// * Действие: другие сообщения об ошибках (если они есть) подскажут
//             причину выдачи этого сообщения.
```

Эта ошибка обычно возникает по одной из следующих причин:

- в имени внешней процедуры в оболочке PL/SQL или в коде на языке C сделана ошибка;
- разработчик забыл экспортировать функцию в Windows **(_declspec(dllexport)).**

Для демонстрации этой ошибки я изменил исходный код в файле **lobtofile.pc** следующим образом:

```
tifdef WIN_NT
_declspec  (dllexport)
#endif
int xlobToFile(OCIExtProcContext  *  ctx,
               char *                filename,
```

Я добавил x к имени файла. Теперь при попытке выполнения мы получаем:

```
declare x clob; y number; begin y := lob_io.write('x', x); end;
*
ERROR at line 1:
ORA-06521: PL/SQL: Error mapping function
```

```
ORA-06522: ld.so.l: extprocPLSExtProc: fatal: lobToFile: can't find
symbol
OPA-06512: at "OPS$TKYTE.LOB_IO", line 0
ORA-06512: at line 1
```

Это показывает, что причина ошибки — can't find symbol, т.е. нет соответствия между именем, указанным в PL/SQL-коде, и именем функции во внешней библиотеке. Либо вкралась опечатка, либо имя функции не экспортировано (в среде Windows).

ORA-06523 "превышено максимальное количество аргументов"

```
$ oerr ora 6523
06523, 00000, "Maximum number of arguments exceeded"
// * Причина: имеется ограничение на количество аргументов, которые можно
//            передавать внешней функции.
// * Действие: в документации для своей версии сервера и платформы
//            проверьте, как вычисляется максимальное количество
//            аргументов.
```

Это сообщение об ошибке можно получить в случае слишком большого списка параметров. Во внешние процедуры обычно можно передавать до 128 параметров (меньше, если передаются числа двойной точности, double, поскольку они занимают 8 байт, а не 4). При получении этого сообщения об ошибке, если действительно необходимо передавать столько параметров, проще всего обойти это ограничение с помощью набора. Например, следующий фрагмент:

```
 1  declare
 2      l_input   strArray := strArray();
 3      l_output  strArray := strArray();
 4  begin
 5      dbms_output.put_line('Pass strArray');
 6      for i in 1 .. 1000 loop
 7          l_input.extend;
 8         linput(i) := 'Element ' || i;
 9      end loop;
10      demo_passing_pkg.pass(l_input, l_output);
11      dbms_output.put_line('l_input.count = ' || l_input.count ||
12                           ' l_output.count = ' || l_output.count);
13      for i in 1 .. l_input.count loop
14          if (l_input(i) != l_output(i)) then
15              raise program_error;
16          end if;
17      end loop;
18* end;
SQL> /
Pass strArray
l_input.count = 1000 l_output.count = 1000

PL/SQL procedure successfully completed.
```

показывает, что с помощью набора я могу передавать внешней процедуре 1000 строк - во много раз превысив ограничение на количество параметров.

ORA-06525 "неправильная длина для данных типа CHAR или RAW"

```
06525, 00000, "Length Mismatch for CHAR or RAW data"
// * Причина: значение, указанное в переменной, задающей длину строки,
//            недопустимо. Эта ошибка может произойти, если в PL/SQL
//            переменная типа RAW указана в качестве параметра,
//            передаваемого в режиме INOUT, OUT или в качестве
//            возвращаемого значения, но соответствующая переменная,
//            задающая длину, не передана. Эта ошибка может также
//            возникать при несоответствии заданного в переменной
//            значения длины фактической длине данных типа orlvstr или
//            orlraw.
//
// * Действие: исправьте код внешней процедуры и правильно задайте
//             переменную, определяющую длину.
```

Это сообщение об ошибке, если вы следуете моим принципам передачи и возврата параметров, может произойти только в случае возвращения из функции данных типа RAW и строки. Решение проблемы очень простое: надо правильно задать длину. Для пустого параметра типа RAW, переданного в режиме OUT, необходимо установить длину 0, как делалось в представленных выше примерах. Для непустого параметра типа RAW, переданного в режиме OUT, длина должна быть меньше или равна атрибуту MAXLEN. Аналогично, длина возвращаемой строки тоже должна устанавливаться правильно: меньше чем MAXLEN, но, поскольку память для строки выделяет внешняя процедура, значения MAXLEN она не получает, поэтому атрибут LENGTH должен быть меньше или равен 32760 (максимальное значение, которое может быть обработано в PL/SQL).

ORA-06526 "невозможно загрузить библиотеку PL/SQL"

```
$ oerr ora 6526
06526, 00000, "Unable to load PL/SQL library"
// * Причина: PL/SQL не смог загрузить библиотеку, на которую ссылается
//            конструкция EXTERNAL. Это серьезная ошибка, которая обычно
//            не возникает.
//
// * Действие: сообщите о проблеме службе поддержки.
```

Это внутренняя ошибка. Сообщение о ней выдаваться не должно, но если уж оно получено, возможны два варианта.

Во-первых, это сообщение об ошибке может сопровождаться другим, более детальным сообщением. Например:

```
ERROR at line 1: ORA-6526: Unable to load PL/SQL library
ORA-4030: out of process memory when trying to allocate 65036 bytes
 (callheap,KQL tmpbuf)
```

Второе сообщение самоочевидно: не хватает памяти. Необходимо сократить объем используемой сеансом памяти.

Если же сопровождающее ошибку ORA-6526 сообщение не позволяет понять причину ошибки, обращайтесь в службу поддержки.

ORA-06527 "ошибка SQLLIB внешней процедуры: %s"

```
$ oerr ora 6527
06527, 00000, "External procedure SQLLIB error: %s"
// * Причина:  при выполнении внешней процедуры, написанной с помощью
//             Pro*C, возникла ошибка в библиотеке sqllib.
//
// * Действие: тест сообщения позволит понять, какая именно ошибка
//             произошла в библиотеке SQLLIB. Обратитесь к руководству
//             Oracle Error Messages and Codes, где можно найти полное
//             описание причин ошибки, и выполните соответствующие
//             действия.
```

С этой ошибкой все понятно. Более детальная информация о причинах будет представлена в сообщении.

Резюме

В этой главе мы рассмотрели тонкости использования внешних процедур:

- поддержка информации о состоянии с помощью контекстов;
- использование независимых от ОС функций для работы с файлами;
- параметризация кода внешней процедуры с помощью внешних файлов параметров;
- оснащение кода средствами отладки (с помощью макроса **debugf),** позволяющими найти причину проблем;
- приемы написания безопасного кода **(всегда** передавать контекст, **всегда** передавать индикаторы значений **NULL** и т.д.);
- как использовать универсальный шаблон для быстрой разработки внешних процедур с широкими функциональными возможностями;
- различия между внешними процедурами, использующими исключительно библиотеку OCI, и процедурами, использующими средства прекомпилятора Pro*C;
- как сопоставлять, принимать и передавать основные типы данных PL/SQL во внешние функции на языке C;
- как передавать и принимать наборы данных.

Имея представленные выше универсальный шаблон и файлы управления проектом, вы получили все необходимое для написания внешней процедуры от начала до конца за несколько минут. Самое сложное — сопоставление типов данных, но по представленным в этой главе таблицам это легко сделать. Они вам подскажут, какой тип использовать для передачи данных. Далее следуйте представленным в примерах принципам передачи параметров (всегда передавать контекст, всегда передавать атрибут MAXLEN для строк и данных типа **RAW,** всегда передавать индикаторные переменные и т.д.). Это обеспечит создание эффективных внешних процедур в кратчайшие строки.

19

Хранимые процедуры на языке Java

В сервере Oracle 8.1.5 впервые появилась возможность использовать для реализации хранимых процедур язык Java. Для 99 процентов задач всегда хватало возможностей языка PL/SQL, и его по-прежнему можно использовать. В Oracle 8.0 ранее появилась возможность реализовать процедуры на языке C (см. главу 18). Хранимые процедуры на языке Java (еще один вид внешних процедур) — естественное расширение этой возможности, позволяющее использовать язык Java в тех случаях, когда раньше приходилось программировать на C или C++.

Если необходимо разработать хранимую процедуру, теперь есть как минимум три возможности: использовать язык PL/SQL, Java или C. Я перечислил их в порядке предпочтения. Большую часть обработки в базе данных можно выполнить на PL/SQL. Если что-то нельзя сделать с помощью PL/SQL (в основном это касается интерфейсов с ОС), вступает в игру язык Java. Язык C используется при наличии уже созданного кода на C или в тех случаях, когда нельзя решить задачу средствами Java.

Эта глава не раскрывает основы Java, интерфейса JDBC или программирования с помощью SQLJ. Предполагается, что читатель хоть немного знаком с языком Java и сможет разобраться в небольших фрагментах Java-кода. Предполагается также общее знание интерфейса JDBC и прекомпилятора SQLJ, хотя при наличии минимального опыта использования Java вы легко сможете понять фрагменты кода, связанные с JDBC и SQLJ.

Когда используются хранимые процедуры на языке Java?

Внешние процедуры на языке Java отличаются от процедур на C тем, что, как и программные единицы PL/SQL, они выполняются встроенной виртуальной Java-машиной (JVM) сервера Oracle, непосредственно в адресном пространстве сервера. Чтобы использовать внешние процедуры на языке C, необходимо сконфигурировать процесс прослушивания, настроить файл **TNSNAMES.ORA** и запустить отдельный процесс. При использовании языка Java все это не нужно, поскольку как интерпретируемый язык он считается "безопасным" (как и PL/SQL). Нельзя создать Java-функцию, переписывающую часть области SGA. Это и хорошо, и плохо, как выяснится по ходу обсуждения. Тот факт, что работа происходит в одном адресном пространстве, обеспечивает более быстрое взаимодействие между кодом на Java и сервером, в частности происходит меньше переключений контекста между процессами на уровне ОС. С другой стороны, однако, Java-код всегда работает с правами "владельца ПО Oracle", поэтому хранимая процедура на Java (при наличии соответствующих привилегий) может переписать файл параметров инициализации сервера, **INIT.ORA** (или другие, еще более важные файлы, например файлы данных).

Лично я постоянно использую небольшие фрагменты Java-кода для реализации того, что невозможно сделать с помощью PL/SQL. Например, в приложении A, посвященном основным стандартным пакетам, я демонстрирую, как я реализовал пакет для работы с сокетами TCP/IP при помощи Java. Я создавал его для версии Oracle 8.1.5, до появления пакета **UTL_TCP** (который тоже написан на языке Java), и предпочитаю использовать его до сих пор. Я также использую средства языка Java для передачи сообщений электронной почты с сервера. И для этих целей уже существует стандартный пакет, **UTL_SMTP** (тоже реализованный на языке Java), позволяющий отправлять простые сообщения, но непосредственное использование языка Java открывает множество других возможностей, включая передачу (и получение) сообщений электронной почты с вложениями.

Я интенсивно использую пакет **UTL_FILE** для чтения и записи файлов в **PL/SQL.** Одна из возможностей, которых не хватает пакету **UTL_FILE,** — получение списка файлов в каталоге. С помощью языка PL/SQL его получить нельзя, а на Java — элементарно.

Иногда необходимо выполнить команду ОС или программу из среды сервера. В этом случае язык PL/SQL тоже не поможет, а Java позволит легко решить задачу. Изредка мне необходимо узнать часовой пояс, установленный на сервере. В PL/SQL его получить нельзя, а вот с помощью Java — можно (эту возможность мы рассмотрим в приложении A при изучении стандартного пакета **UTL_TCP).** Надо измерять время с точностью до миллисекунд? В Oracle 8i с помощью Java это можно сделать.

Если постоянно необходимо подключаться к СУБД DB2 для выполнения запросов, это можно сделать с помощью шлюза (Transparent Gateway) для СУБД DB2. Это позволит без ограничений выполнять соединения таблиц в разнородных базах данных, распределенные транзакции, прозрачную двухэтапную фиксацию и использовать много других возможностей. Но если необходимо выполнить запрос или изменение в базе дан-

ных DB2 и все перечисленные потрясающие возможности не нужны, достаточно загрузить в базу данных драйверы JDBC для DB2, написанные на языке Java, и воспользоваться ими (естественно, это применимо не только для СУБД DB2).

По сути, любой из миллионов имеющихся не интерактивных (не обладающих пользовательским интерфейсом) фрагментов Java-кода можно загрузить в базу данных Oracle и использовать. Вот почему фрагменты Java-кода постоянно встречаются в приложениях.

Я предпочитаю использовать язык Java, только когда это удобно и необходимо. Я по-прежнему считаю PL/SQL подходящим средством для создания подавляющего большинства хранимых процедур. Написав одну-две строки PL/SQL-кода, можно получить тот же результат, что и в случае многострочной программы на Java/JDBC. Препроцессор SQLJ уменьшает объем необходимого кода, но выдаваемый им код по производительности уступает сочетанию языков PL/SQL и SQL. Производительность кода на PL/SQL при взаимодействии с SQL выше, чем для сочетания Java/JDBC, как и можно было предположить. Язык PL/SQL проектировался как расширение SQL, и они очень тесно интегрированы. Большинство типов данных языка PL/SQL — это стандартные типы данных SQL, а все типы данных SQL включены в PL/SQL. Между этими языками нет несоответствия типов. Доступ к SQL из кода на Java выполняется средствами функционального интерфейса, добавленного к языку. Каждый тип данных SQL необходимо преобразовать в некий тип данных Java, и, наоборот, все SQL-операторы выполняются процедурно, т.е. между этими языками нет тесной связи. Итак, если выполняется обработка данных в базе, надо использовать язык PL/SQL. Если надо на время выйти за пределы базы данных (например, чтобы отправить сообщение по электронной почте), лучшим средством для этого является язык Java. Если необходимо выполнить поиск в сообщениях электронной почты, хранящихся в базе данных, используйте язык PL/SQL. Если же необходимо загрузить сообщения электронной почты в базу данных, используйте Java.

Как работают внешние процедуры на языке Java

Оказывается, внешние процедуры на языке Java (термин "внешняя процедура" в данном случае является синонимом "хранимой процедуры") создавать значительно проще, чем на языке C. Например, в предыдущей главе, посвященной созданию внешних процедур на языке C, пришлось решать следующие проблемы.

- **Управление состоянием.** Внешние процедуры могут потерять информацию о состоянии (текущие значения статических или глобальных переменных). Это связано с используемым механизмом кэширования динамически подключаемых библиотек. Поэтому необходим механизм определения и сохранения состояния в программах на языке C.
- **Механизмы трассировки.** Внешние процедуры выполняются на сервере как **отдельный** процесс. Хотя на некоторых платформах эти процедуры можно отлаживать с помощью обычного отладчика, это весьма сложно и, если ошибки возни-

кают только при одновременном использовании внешней процедуры большим количеством пользователей, просто невозможно. Необходимо средство генерации трассировочных файлов по требованию, "начиная с этого момента".

- **Использование параметров.** Необходимо средство параметризации внешних процедур, чтобы можно было управлять их работой извне с помощью файла параметров, аналогично тому, как файл **init.ora** используется для управления сервером.
- **Общая обработка ошибок.** Необходимо простое средство выдачи пользователю вразумительных сообщений об ошибках.

При использовании языка Java оказывается, что управление состоянием, трассировка и общая обработка ошибок уже не является проблемой. Для сохранения информации о состоянии достаточно объявить переменные в создаваемых Java-классах. Для обеспечении простейшей трассировки можно использовать вызовы **System.out.println.** Общую обработку ошибок можно выполнять с помощью функции **RAISE_APPLICATION_ERROR** языка PL/SQL. Все это продемонстрировано в следующем коде:

```
tkyte@TKYTE816>create or replace and compile
  2  java source named "demo"
  3  as
  4 import java.sql.SQLException;
  5
  6 public class demo extends Object
  7  {
  8
  9  static int counter = 0;
 10
 11 public static int IncrementCounter() throws SQLException
 12  {
 13   System.out.println("Входим   в функцию IncrementCounter,
-> counter = "+counter);
 14      if (++counter >= 3)
 15      {
 16       System.out.println("Ошибка      counter="+counter);
 17          #sql {
 18         begin raise_application_error(-20001,
-> 'Слишком много вызовов'); end;
 19          };
 20      }
 21   System.out.println("Выходим   из функции IncrementCounter,
-> counter = "+counter);
 22      return counter;
 23  }
 24  }
 25  /

Java created.
```

Состояние поддерживается с помощью статической переменной **counter**. Наша простая демонстрационная программа будет увеличивать счетчик при каждом вызове, а

начиная с третьего и при последующих вызовах, будет автоматически выдавать сообщение об ошибке.

Обратите внимание, что для создания небольших фрагментов кода вроде этого можно использовать утилиту SQL*Plus, непосредственно загружающую Java-код в базу данных, автоматически компилируя его в байт-код и запоминая в соответствующих структурах. Ни внешний компилятор, ни средства разработки JDK при этом не нужны — достаточно SQL-оператора **CREATE OR REPLACE**. Именно так я и предпочитаю создавать хранимые процедуры на языке Java. Это упрощает их установку на любой платформе. Не нужно запрашивать имя пользователя и пароль, как при использовании команды **LOADJAVA** (это утилита командной строки для загрузки исходного кода, классов Java или jar-файлов в базу данных). Не надо думать о каталогах для поиска классов (classpath) и других подобных нюансах. В приложении А мы рассмотрим утилиту **LOADJAVA** и пакет **DBMS_JAVA**, обеспечивающий интерфейс к программе **LOADJAVA**.

Этот метод (с использованием оператора **CREATE OR REPLACE**) загрузки небольших Java-функций в базу данных особенно хорошо подходит для тех, кто только начинает осваивать технологии Java. Вместо установки JDBC-драйверов, среды разработки JDK, настройки списка каталогов для поиска классов можно просто компилировать код в базе данных, точно так же, как при создании программных единиц PL/SQL. Сообщения об ошибках компиляции выдаются точно так же, как и при использовании языка PL/SQL, например:

```
tkyte@TKYTE816>create or replace and compile
  2  java source named "demo2"
  3  as
  4
  5  public class demo2 extends Object
  6   {
  7
  8  public static int my_routine()
  9   {
 10    System.out.println("Входим    в функцию my_routine");
 11
 12      return counter;
 13   }
 14   }
 15   /

Warning: Java created with compilation errors.

tkyte@TKYTE816>show errors java source "demo2"
Errors for JAVA SOURCE demo2:

LINE/COL ERROR

0/0     demo2:8: Undefined variable: counter
0/0       Info: 1 errors
```

Это показывает, что функция **my_routine**, определенная в строке 8, обращается к необъявленной переменной. Не приходится выискивать ошибку в коде, поскольку получено информативное сообщение о ней. Я не раз убеждался, что многократных оши-

бок при настройке JDBC/JDK/CLASSPATH можно легко избежать, загрузив за пару секунд код с помощью этого простого подхода.

Вернемся теперь к работающему примеру. Хочу обратить ваше внимание на еще одну важную деталь в созданном выше классе. Метод, вызываемый из языка SQL, **IncrementCounter**, объявлен как статический. Он обязательно должен быть статическим. (Хотя не все должно быть статическим: при реализации статического метода можно использовать обычные методы). Для языка SQL необходим хотя бы один метод, который можно вызвать, не передавая неявно данные экземпляра с помощью скрытого параметра, вот почему нужен статический метод.

Теперь, после загрузки небольшого Java-класса, необходимо создать для него спецификацию вызова в языке PL/SQL. Эта процедура очень похожа на ту, что была описана в главе 18 для внешних процедур на языке C, когда мы сопоставляли типы данных C типам данных SQL. Именно это мы и сделаем сейчас; только на этот раз будут сопоставляться типы данных языка Java типам данных SQL:

```
tkyte@TKYTE816>create or replace
  2  function java_counter return number
  3  as
  4  language java
  5  name 'demo.IncrementCounter() return integer';
  6  /

Function created.
```

Теперь можно вызывать эту функцию:

```
tkyte@TKYTE816>set serveroutput on

tkyte@TKYTE816> exec dbms_output.put_line(java_counter);
1
PL/SL procedure successfully completed.

tkyte@TKYTE816> exec dbms_output.put_line(java_counter);
2
PL/SL procedure successfully completed.

tkyte@TKYTE816> exec dbms_output.put_line(java_counter);
BEGIN dbms_output.put_line(java_counter); END;
*

ERROR at line 1:
ORA-29532: Java call terminated by uncaught Java exception:
oracle.jdbc.driver.OracleSQLException:
ORA-20001: Слишком иного вызовов
ORA-06512: at line 1
ORA-06512: at "TKYTE.JAVA_COUNTER", line 0
ORA-06512: at line 1
```

Как видите, информация о состоянии поддерживается автоматически, о чем свидетельствует увеличение счетчика с 1 до 2 и 3. Об ошибках сообщать тоже достаточно легко, но куда попадают результаты обращения к System.out.println? По умолчанию они попадают в файл трассировки. При наличии доступа к представлениям **V$PROCESS**, **V$SESSION** и **V$PARAMETER** можно определить имя файла трассировки в конфигура-

ции выделенного сервера следующим образом (этот пример предназначен для Windows - для ОС UNIX он будет аналогичным, но полученное имя файла будет другим):

```
tkyte@TKYTE816>select  c.value||'\ORA'||to_char(a.spid,'fm00000')||•.trc'
  2        from v$process a,  v$session b,  v$parameter c
  3       where a.addr = b.paddr
  4         and b.audsid = userenv('sessionid')
  5         and c.name =  'user_dump_dest'
  6  /

C.VALUE||'\ORA'||TO_CHAR(A.SPID,'FM00000')||'.TRC'

C:\oracle\admin\tkyte816\udump\ORA01236.trc
tkyte@TKYTE816> edit C:\oracle\admin\tkyte816\udump\ORA01236.trc
```

В этом файле можно обнаружить следующее:

```
Dump file C:\oracle\admin\tkyte816\udump\ORA01236.TRC
Tue Mar 27 11:15:48 2001
ORACLE V8.1.6.0.0 - Production vsnsta=0
vsnsql=e vsnxtr=3
Windows 2000 Version 5.0 , CPU type 586
Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option
JServer Release 8.1.6.0.0 — Production
Windows 2000 Version 5.0 , CPU type 586
Instance name: tkyte816
Redo thread mounted by this instance: 1
Oracle process number: 12
Windows thread id: 1236, image: ORACLE.EXE

*** 2001-03-27 11:15:48.820
*** SESSION ID:(8.11) 2001-03-27 11:15:48.810
Входим в функцию IncrementCounter, counter = 0
Выходим из функции IncrementCounter, counter = 1
Входим в функцию IncrementCounter, counter = 1
Выходим из функции IncrementCounter, counter = 2
Входим в функцию IncrementCounter, counter = 2
Ошибка! counter=3
oracle.jdbc.driver.OracleSQLException: ORA-20001: Слишком много вызовов
ORA-06512: at line 1
```

Я также мог бы использовать средства пакета **DBMS_JAVA** для перенаправления этих результатов на экран утилиты SQL*Plus, чтобы избежать поиска соответствующего трассировочного файла при отладке функции. В этом разделе периодически упоминается пакет **DBMS_JAVA**, но полное его описание будет представлено в соответствующем разделе приложения А.

Из этого небольшого примера понятно, что, по сравнению с созданием внешних процедур на языке С, создавать хранимые процедуры на Java — просто. Не нужно специально настраивать сервер — только инсталлировать Java в базу данных. Не нужен внешний компилятор. Многие средства, которые в случае языка С пришлось создавать самим, мы получаем от сервера автоматически. Это на самом деле просто.

Я не описывал пока конфигурирование Java-кода с помощью файла параметров. Причина в том, что Java содержит встроенные средства для этого в виде класса **java.util.Properties.** Достаточно использовать метод **load** этого класса для загрузки ранее сохраненного набора свойств либо из большого объекта в таблице базы данных, либо из файла ОС, — что больше подходит.

Далее я представлю несколько полезных примеров хранимых процедур на языке Java, в частности, упоминавшихся ранее в разделе "Когда используются хранимые процедуры на языке Java?". Но до этого я хочу переписать представленный в главе 18 пакет **DEMO_PASSING_PKG** на языке Java вместо C, чтобы продемонстрировать, как передавать и принимать основные типы данных SQL во внешних процедурах на языке Java.

Передача данных

В этом примере я собираюсь создать ряд процедур с параметром, передаваемым в режиме **IN,** и параметром, передаваемым в режиме OUT (или **IN** OUT). Мы напишем по процедуре для каждого из интересующих нас типов данных (наиболее часто используемых). При этом будет продемонстрирован правильный способ передачи входных данных и получения результатов каждого типа. Кроме того, я создам несколько функций и покажу, как возвращать данные некоторых из этих типов. Меня при работе с Java интересуют следующие типы:

- строки (размером до 32 Кбайт);
- числа (произвольного масштаба и точности);
- даты;
- целые числа (включая данные типа **binary_integer);**
- данные типа **RAW** (размером до 32 Кбайт);
- большие объекты (для любых данных размером более 32 Кбайт);
- массивы строк;
- массивы чисел;
- массивы дат.

Этот список несколько отличается от аналогичного списка для внешних процедур на языке C. В частности, в нем не указан тип данных **BOOLEAN.** Дело в том, что пока нет соответствия между типом данных **PL/SQL BOOLEAN** и типами данных языка Java. Нельзя передавать данные типа **BOOLEAN** как параметры внешним процедурам, написанным на языке Java.

С помощью объектно-реляционных расширений можно создавать типы данных любой сложности. Для создания таких типов данных я рекомендую использовать поставляемое корпорацией Oracle Java-средство JPublisher. Оно автоматически создает Java-классы, соответствующие объектным типам. Подробнее о JPublisher можно почитать в руководстве *Oracle8i JPublisher User's Guide,* которое входит в набор документации, предлагаемой корпорацией Oracle. Как и в случае внешних процедур на языке C, мы не будем углубляться в особенности использования объектных типов в хранимых процедурах на Java, ограничившись только простыми наборами данных скалярных типов.

Java-класс будет создан для тех же целей, что и представленная в предыдущей главе динамически подключаемая библиотека на языке C. Начнем с SQL-операторов для создания трех типов наборов — они совпадают с использовавшимися в примередля языка C в предыдущей главе:

```
tkyte@TKYTE816>create or replace type numArray as table of number;
Type created.

tkyte@TKYTE816>create or replace type dateArray as table of date;
Type created.

tkyte@TKYTE816> create or replace type strArray as table of varchar2(255);
Type created.
```

Теперь рассмотрим спецификацию PL/SQL-пакета для этого примера. Она будет состоять из набора перегруженных процедур и функций для проверки приема и передачи параметров в хранимых процедурах на языке Java. Каждая подпрограмма имеет параметр, предаваемый в режиме IN, и параметр, передаваемый в режиме **OUT,** что позволяет продемонстрировать передачу данных в Java-код и возвращение результатов.

Первая процедура передает числовые данные. Данные Oracle типа **NUMBER** будут передаваться как Java-тип **BigDecimal.** Их можно принимать и как данные типа **int,** и как строки и как другие типы, но при этом возможна потеря точности. Данные типа **BigDecimal** могут без проблем принять любое значение типа **NUMBER** от сервера Oracle.

Обратите внимание, что параметр, передаваемый в режиме **OUT,** на уровне Java принимается как массив данных типа **BigDecimal.** Так будет для всех параметров, передаваемых Java в режиме OUT. Для изменения параметра, переданного Java, нужно передавать "массив" параметров (в этом массиве будет только один элемент) и изменять соответствующий элемент массива. Далее, при описании кода на языке Java, вы увидите, как это сказывается на исходном коде.

```
tkyte@TKYTE816>create or replace package demo_passing_pkg
  2  as
  3     procedure pass(p_in in number, p_out out number)
  4       as
  5       language java
  6       name 'demo_passing_pkg.pass(java.math.BigDecimal,
  7                                   java.math.BigDecimal[])'
```

Даты Oracle сопоставляются типу данных **Timestamp**. И в этом случае можно было бы сопоставить датам множество других типов, например **String**, но во избежание потери информации при неявных преобразованиях я выбрал тип **Timestamp**, который позволяет сохранить все данные, содержащиеся в объектах Oracle типа **DATE**.

```
  8
  9     procedure pass(p_in in date, p_out out date)
 10       as
 11      language java
 12     name 'demo_passing_pkg.pass(java.sql.Timestamp,
 13                                 java.sql.Timestamp[])';
```

Строки типа VARCHAR2 передаются очень просто — как данные типа **java.lang.String.**

```
14
15      procedure pass(p_in in varchar2, p_out out varchar2)
16      as
17      language java
18      name 'demo_passing_pkg.pass(java.lang.String,
19                                  java.lang.String[])';
```

Для данных типа **CLOB** мы используем предоставляемый Oracle Java-тип **oracle.sql.CLOB.** С помощью этого типа мы сможем получить входной и выходной потоки данных, используемые для чтения и записи данных типа **CLOB.**

```
20
21      procedure pass(p_in in CLOB, p_out in out CLOB)
22      as
23      language java
24      name 'demo_passing_pkg.pass(oracle.sql.CLOB,
25                                  oracle.sql.CLOB[])';
```

Теперь перейдем к наборам: вы видели, что, независимо от типа фактически передаваемого набора, используется один и тот же предоставляемый Oracle тип. Вот почему в данном случае Java-функции не являются перегруженными, как все предыдущие (пока что все вызываемые Java-функции назывались demo_passing_pkg.pass). Поскольку все типы наборов передаются как один и тот же тип Java, перегрузку имен использовать нельзя — необходимо называть функцию в соответствии с реально передаваемым типом данных:

```
26
27      procedure pass(p_in in numArray, p_out out numArray)
28      as
29      language java
30      name 'demo_passing_pkg.pass_num_array(oracle.sql.ARRAY,
31                                            oracle.sql.ARRAY[])';
32
33      procedure pass(p_in in dateArray, p_out out dateArray)
34      as
35      language java
36      name 'demo_passing_pkg.pass_date_array(oracle.sql.ARRAY,
37                                             oracle.sql.ARRAY[])';
38
39      procedure pass(p_in in strArray, p_out out strArray)
40      as
41      language java
42      name 'demo_passing_pkg.pass_str_array(oracle.sql.ARRAY,
43                                            oracle.sql.ARRAY[])';
```

Следующие две процедуры демонстрируют сопоставление для типов RAW и INT. SQL-тип RAW будет сопоставляться встроенному типу byte языка Java. Для целых чисел будет использоваться встроенный тип данных int языка Java:

```
44
45      procedure pass_raw(p_in in RAW, p_out out RAW>
46      as
47      language java
```

```
48     name 'demo_passing_pkg.pass(byte[], byte[][])';
49
50      procedure pass_int(p_in   in number,
51                         p_out  out number)
52      as
53      language java
54       name 'demo_passing_pkg.pass_int(int, int[])';
```

Наконец, для полноты изложения я продемонстрирую использование функций для возвращения данных простых скалярных типов:

```
55
56      function return_number return number
57     as
58      language java
59      name 'demo_passing_pkg.return_num() return java.math.BigDecimal';
60
61      function return_date return date
62     as
63      language java
64      name 'demo_passing_pkg.return_date() return java.sql.Timestamp';
65
66      function return_string return varchar2
67     as
68      language java
69      name 'demo_passing_pkg.return_string() return java.lang.String';
70
71  end demo_passing_pkg;
72  /

Package created.
```

Эта спецификация пакета практически совпадает (за исключением процедур для данных типа BOOLEAN) с той, что использовалась для внешних процедур на языке C. В этом примере я поместил уровень связывания непосредственно в спецификацию, чтобы не пришлось писать избыточное тело пакета (все функции написаны на языке Java).

Рассмотрим Java-код, реализующий использованные выше функции. Начнем с определения Java-класса demo_passing_pkg:

```
tkyte@TKYTE816> set define off

tkyte@TKYTE816> create or replace and compile
  2  java source named "demo_passing_pkg"
  3   as
  4  import java.io.*;
  5  import java.sql.*;
  6  import java.math.*;
  7  import oracle.sql.*;
  8  import oracle.jdbc.driver.*;
  9
 10  public class demo_passing_pkg extends Object
 11   {
```

В первом из представленных далее методов демонстрируется единственно возможный способ передачи параметров в режиме **OUT** функции на Java; фактически мы передаем массив из одного элемента. При изменении значения в массиве изменяется параметр, переданный в режиме **OUT.** Вот почему все эти методы в качестве второго параметра принимают массив. Значение **p_out[0]** можно изменять, и оно будет передано методом в вызывающую среду. Изменения значения **p_in** в вызывающую среду не передаются.

Интересная особенность данного метода — отсутствие индикаторной переменной. Язык Java поддерживает понятие пустого объекта **(null)** в объектных типах, как и языки SQL и PL/SQL. Он, однако, не поддерживает трехзначную логику, как SQL; операции **X IS NOT NULL** нет — можно только непосредственно сравнивать объект с **null.** Не перепутайте и не пытайтесь писать условия вида **p_in <> NULL** в PL/SQL-коде, поскольку они не будут работать корректно.

```
12 public static void pass(java.math.BigDecimal p_in,
13                         java.math.BigDecimal[] p_out)
14  {
15     if (p_in != null)
16      {
17         System.out.println
18         ("Первый параметр " + p_in.toString());
19
20         p_out[0] = p_in.negate();
21
22         System.out.println
23        ("Устанавливаем параметр out равным " + p_out[0].toString());
24      }
25  }
```

Следующий метод работает с типом данных Oracle **DATE**. Он совпадает с представленным выше, за исключением того, что используются методы класса **Timestamp** для обработки даты. Наша задача — добавить к переданной дате один месяц:

```
26
27 public static void pass(java.sql.Timestamp p_in,
28                         java.sql.Tiraestamp[] p_out)
29  {
30     if (p_in != null)
31      {
32         System.out.println
33         ("Первый параметр " + p_in.toString());
34
35         p_out[0] = p_in;
36
37         if (p_out[0].getMonth() < 11)
38            p_out[0].setMonth(p_out[0].getMonth()+1);
39          else
40          {
41              p_out[0].setMonth(0);
42            p_out[0].setYear(p_out[0].getYear()+1);
43          }
```

```
44          System.out.println
45          ("Устанавливаем параметр out равным " + p_out[0].toString());
46      }
47  }
```

Теперь переходим к самому простому из типов данных — **String**, который соответствует строковым типам SQL. Если вспомнить версию на языке C, с шестью формальными параметрами, индикаторными переменными, атрибутами strlen, функциями strcpy и т.п., то по сравнению с ней эта реализация тривиальна:

```
48
49  public static void pass(java.lang.String p_in,
50                          java.lang.String[] p_out)
51  {
52      if (p_in != null)
53      {
54          System.out.println
55          ("Первый параметр " + p_in.toString());
56
57          p_out[0] = p_in.toUpperCase();
58
59          System.out.println
60          ("Устанавливаем параметр out равным " + p_out[0].toString());
61      }
62  }
```

В методе для данных типа **CLOB** придется выполнить ряд дополнительных действий. Для того чтобы показать, как принимать и возвращать большие объекты, здесь выполняется копирование. Вы видите, что для изменения/чтения содержимого большого объекта используются стандартные потоки чтения/записи языка Java. В этом примере is — входной поток, а **os** — выходной. Метод копирует данные фрагментами по 8 Кбайт. Выполняется цикл чтения и записи, пока не закончатся считываемые данные:

```
63
64  public static void pass(oracle.sql.CLOB p_in,
65                          oracle.sql.CLOB[] p_out)
66  throws SQLException, IOException
67  {
68      if (p_in != null && p_out[0] != null)
69      {
70          System.out.println
71          ("Первый параметр " + p_in.length());
72          System.out.println
73          ("Первый параметр '" +
74          p_in.getSubString(1,80) + " ' " ) ;
75
76          Reader is = p_in.getCharacterStream();
77          Writer os = p_out[0].getCharacterOutputStream();
78
79          char buffer!] = new char[8192];
80          int length;
81
```

```
82        while((length-is.read(buffer,0,8192)) != -1)
83              os.write(buffer,0,length);
84
85          is.close();
86          os.close();
87
88         System.out.println
89         ("Устанавливаем параметр out равным " +
90          P_out[0].getSubString(1,80));
91      }
92  }
```

Следующий метод — приватный (внутренний). Он выдает метаданные о переданном ему объекте типа **oracle.sql.ARRAY**. Для каждого из передаваемых Java трех типов массивов будет вызываться этот метод, информирующий о том, какого размера и типа массив передан:

```
93
94 private static void show_array_info(oracle.sql.ARRAY p_in)
95 throws SQLException
96 {
97     System.out.println("Тип массива      " +
98                         p_in.getSQLTypeName());
99    System.out.println("Код типа массива " +
100                       p_in.getBaseType()) ;
101    System.out.println("Длина массива    " +
102                        p_in.length());
103 }
```

Теперь рассмотрим методы для обработки этих массивов. Использовать массивы несложно, если разобраться, как получать из них данные и изменять их. Получить данные очень просто; метод **getArray()** возвращает базовый массив данных. Приведя возвращаемое методом **getArray()** значение к нужному типу, мы получим Java-массив этого типа. Поместить данные в такой массив немного сложнее. Необходимо сначала получить дескриптор (метаданные) массива, а затем создать новый объект-массив с этим дескриптором и соответствующими значениями. Следующий набор методов продемонстрирует это для каждого из использованных типов массивов. Обратите внимание, что тексты методов практически совпадают, за исключением фактических обращений к массивам данных Java. Эти методы выдают метаданные для типа *oracle.sql.ARRAY*, выдают содержимое массива и копируют входной массив в выходной:

```
104
105 public static void pass_num_array(oracle.sql.ARRAY p_in,
106                                    oracle.sql.ARRAY[] p_out)
107 throws SQLException
108 {
109     show_array_info(p_in);
110  java.math.BigDecimal[]  values = (BigDecimal[])p_in.getArray();
111
112     for(int i = 0; i < p_in.length(); i++)
113       System.out.println("p_in["+i+"] = " + values[i].toString());
```

```
114
115      Connection conn = new OracleDriver().defaultConnectionO;
116      ArrayDescriptor descriptor =
117         ArrayDescriptor.createDescriptor(p_in.getSQLTypeNatne(),
-> conn);
118
119     p_out[0] = new ARRAY(descriptor, conn, values);
120
121  )
122
123  public static void
124  pass_date_array(oracle.sql.ARRAY p_in, oracle.sql.ARRAY[] p_out)
125  throws SQLException
126  {
127      show_array_info(p_in);
128      java.sql.Timestamp[] values = (Timestamp[])p_in.getArray();
129
130      for(int i = 0; i < p_in.length(); i++)
131      System.out.println("p_in["+i+"]     = " + values[i].toString());
132
133      Connection conn = new OracleDriver().defaultConnection();
134      ArrayDescriptor descriptor =
135         ArrayDescriptor.createDescriptor (p_in.getSQLTypeName(), conn) ;
136
137     p_out[0] = new ARRAY(descriptor, conn, values);
138
139  )
140
141  public static void
142  pass_str_array(oracle.sql.ARRAY p_in, oracle.sql.ARRAY[] p_out)
143  throws java.sql.SQLException,IOException
144  {
145      show_array_info(p_in);
146    String[]    values = (String[])p_in.getArray();
147
148     for(int i = 0; i < p_in.length(); i++)
149         System.out.println("p_in["+i+"] = " + values[i]);
150
151      Connection conn = new OracleDriver().defaultConnection();
152      ArrayDescriptor descriptor =
153        ArrayDescriptor.createDescriptor(p_in.getSQLTypeName(), conn) ;
154
155     p_out[0] = new ARRAY(descriptor, conn, values);
156
157  )
```

Передача данных типа RAW ничем не отличается от передачи строк. С этим типом данных работать очень легко:

```
158
159  public static void pass(byte[] p_in, byte[][] p_out)
160  {
```

```
161     if (p_in != null)
162         p_out[0] = p_in;
163 }
```

Передача целых чисел — **проблематична,** я вообще не рекомендую их передавать. Нет способа передать значение **NULL** — соответствующий тип данных **int** относится к базовым типам данных языка Java. Эти данные не являются объектами и поэтому не могут быть пустыми. Поскольку индикаторные переменные не поддерживаются, то при необходимости обработать пустые значения придется передавать отдельный параметр, а в PL/SQL-коде — проверять соответствующий флаг, чтобы определить, не возвращено ли пустое значение. Соответствующий метод представлен здесь для полноты, но лучше вообще не использовать данные целого типа, особенно как параметры, передаваемые в режиме IN, — Java-метод не сможет определить, что значение не нужно читать, поскольку пустые значения не поддерживаются.

```
164
165 public static void pass_int(int p_in, int[] p_out)
166 {
167     System.out.println
168      ("Входной параметр " + p_in);
169
170     p_out[0] = p_in;
171
172     System.out.println
173     ("Выходной параметр " + p_out[0]);
174 }
```

Наконец, перейдем к функциям. Если помните, на языке C написать их было непросто. Необходимо было выделять память, обрабатывать пустые значения, явно преобразовывать типы данных C в типы данных Oracle и т.д. Каждая C-функция при этом состояла как минимум из десятка строк кода. В случае же Java достаточно добавить оператор **return**:

```
175
176 public static String return_string()
177 {
178     return "Hello World";
179 }
180
181 public static java.sql.Timestamp return_date()
182 {
183      return new java.sql.Timestamp(0);
184 }
185
186 public static java.math.BigDecimal return_num()
187 {
188    return new java.math.BigDecimal("44.3543");
189 }
190
191 }
192 /
```

```
Java created
tkyte@TKYTE816> set define on
```

Запрограммировать функцию на Java гораздо проще, чем на языке C, благодаря тому, что Java выполняет много действий автоматически, "за кадром". Для обеспечения аналогичной функциональности на языке C потребовалось около 1000 строк кода. Выделение памяти, которое требует столько внимания при программировании на C, в случае Java — не проблема. В случае ошибки возбуждается исключительная ситуация. Индикаторные переменные, с которыми надо было возиться в языке C, вообще не нужны в Java. Проблема возникает при передаче типов данных, соответствующих не объектным типам Java, но, как я уже говорил, не следует их использовать, если может потребоваться передать пустые значения.

Поскольку все компоненты созданы, можно вызывать подпрограммы. Например:

```
tkyte@TKYTE816> set serveroutput on size 1000000
tkyte@TKYTE816> exec dbms_java.set_output(1000000)

tkyte@TKYTE816> declare
  2    l_in   strArray := StrArray();
  3     l_out strArray := strArray();
  4  begin
  5     for i in 1 .. 5 loop
  6          l_in.extend;
  7         l_in(i) := 'Элемент ' || i;
  8      end loop;
  9
 10      demo_passing_pkg.pass(l_in, l_out);
 11     for i in 1 .. l_out.count loop
 12         dbms_output.put_line('l_out(' || i || ') = ' || l_out(i));
 13      end loop;
 14  end;
 15  /
Тип массива     SECOND.STRARRAY
Код типа массива 12
Длина массива    5
p_in[0] = Элемент 1
p_in[1] = Элемент 2
p_in[2] - Элемент 3
p_in[3] = Элемент 4
p_in[4] = Элемент 5
l_out(1) = Элемент 1
l_out(2) = Элемент 2
l_out(3) = Элемент 3
l_out(4) = Элемент 4
l_out(5) = Элемент 5

PL/SQL procedure successfully completed.
```

Первые восемь строк результата были сгенерированы Java-методом, а последние пять — PL/SQL-кодом. Значит, мы передали массив из PL/SQL в Java и получили его обратно. С помощью Java-метода мы скопировали входной массив в выходной после распечатки метаданных и значений элементов массива.

Полезные примеры

Я свято верю, что, если задачу можно решить с помощью одного SQL-оператора, это надо делать. Никогда не используйте, например, цикл **FOR** по курсору, если достаточно выполнить оператор **UPDATE**. Если задачу нельзя решить в SQL, попытайтесь решить ее в PL/SQL. Никогда не пишите внешнюю процедуру на языке Java или C, разве что задачу нельзя решить в PL/SQL или реализация на языке C существенно повышает производительность. Если по техническим причинам задачу нельзя решить с помощью PL/SQL, попробуйте решить ее на языке Java. Однако использование Java требует дополнительных ресурсов — памяти, процессорного времени и времени на первоначальный запуск виртуальной машины JVM. Использование PL/SQL также требует дополнительных ресурсов, но они уже выделены, ничего дополнительно запускать не надо.

Теме не менее ряд задач нельзя решить с помощью языка PL/SQL, а при использовании Java они решаются элементарно. Ниже представлены полезные фрагменты Java-кода, используемые мной в повседневной практике. Это, конечно, не исчерпывающий список, а лишь вершина айсберга. В приложении A примеры использования языка Java в Oracle рассмотрены более широко.

Генерация списка файлов каталога

Пакет **UTL_FILE**, который мы уже несколько раз использовали по ходу изложения, хорошо справляется с чтением и записью текстовых файлов. Очень часто, однако, необходимо обработать все файлы в указанном каталоге. Этого пакет не позволяет сделать. Для получения списков файлов каталога нет встроенных методов ни в SQL, ни в PL/SQL. На Java его очень легко получить. Вот как это делается:

```
tkyte@TKYTE816> create global temporary table DIR_LIST
  2  (filename varchar2(255))
  3  on commit delete rows
  4  /
Table created.
```

В этой реализации я решил использовать для возвращения результатов из хранимой процедуры на Java временную таблицу. Я считаю этот метод наиболее удобным, потому что он позволяет в дальнейшем легко сортировать список и выбирать файлы с нужными именами.

Необходим следующий фрагмент Java-кода:

```
tkyte@TKYTE816> create or replace
  2    and compile java source named "DirList"
  3  as
  4  import java.io.*;
  5  import java.sql.*;
  6
  7  public class DirList
  8  {
  9  public static void getList(String directory)
 10                      throws SQLException
 11  {
```

```
 12      File path = new File(directory);
 13    String[]   list = path.list();
 14      String element;
 15
 16     for(int i = 0; i < list.length; i++)
 17     {
 18         element = list[i];
 19        #sql { INSERT INTO DIR_LIST (FILENAME)
 20              VALUES (:element) };
 21     )
 22  }
 23
 24  )
 25  /

Java created.
```

Я решил использовать SQLJ, чтобы сократить программу. Подключение к базе данных уже выполнено, поэтому реализация с помощью интерфейса JDBC потребовала лишь нескольких дополнительных строк кода. Но с помощью препроцессора SQLJ выполнять SQL-операторы в Java так же просто, как и в PL/SQL. Теперь, конечно же, необходимо создать спецификацию вызова:

```
tkyte@TKYTE816>create or replace
  2 procedure get_dir_list(p_directory in varchar2)
  3  as language java
  4 name 'DirList.getList(java.lang.String)';
  5  /

Procedure created.
```

Прежде чем запускать эту процедуру, следует учесть еще один нюанс. Необходимо предоставить процедуре право делать то, что она должна — читать список файлов каталога. В данном случае я обладаю правами администратора базы данных, поэтому могу предоставить соответствующие привилегии сам себе, но обычно приходится обращаться с соответствующим запросом к администратору. Если помните, во введении к этой главе я писал:

*"... Java-код всегда работает с правами владельца ПО Oracle, поэтому хранимая процедура на Java при предоставлении соответствующих привилегий может переписать файл параметров инициализации сервера, **INIT.ORA** (или другие, еще более важные файлы, например файлы данных)."*

Сервер Oracle защищается от этого следующим образом: для выполнения небезопасных действий необходимо явно получить соответствующую привилегию. Попытавшись использовать эту процедуру до получения необходимых привилегий, мы получим следующее сообщение об ошибке:

```
tkyte@TKYTE816>execget_dir_list('c:\temp');
BEGIN get_dir_list<'c:\temp'); END;
*
ERROR at line 1:
```

```
ORA-29532: Java call terminated by uncaught Java exception:
java.security.AccessControlException:
the Permission (java.io.FilePermission c:\temp read) has not been granted by
dbms_java.grant_permission to
SchemaProtectionDomain(TKYTE|PolicyTableProxy(TKYTE))
ORA-06512: at "TKYTE.GET_DIR_LIST", line 0
ORA-06512: at line 1
```

Поэтому предоставим себе право получать список файлов в соответствующем каталоге:

```
tkyte@TKYTE816> begin
  2              dbms_java.grant_permission
  3            (USER,
  4            'java.io.FilePermission',
  5             'c:\temp',
  6         'read');
  7  end;
  8  /
```

PL/SQL procedure successfully completed.

И можно выполнять процедуру:

```
tkyte@TKYTE816> exec get_dir_list('c:\temp');

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select * from dir_list where rownum < 5;

FILENAME


a.sql
abc.dat
activation
activation8i.zip
```

Соответствующие права доступа определяются спецификацией Java2 Standard Edition (J2SE) и подробно описаны на странице **http://java.sun.eom/j2se/l.3/docs/api/java/security/Permission.html. В** приложении А мы подробно рассмотрим пакет DBMS_JAVA и его использование.

Есть еще один нюанс, который необходимо учитывать. Oracle 8.1.6 — первая версия СУБД Oracle, поддерживающая систему прав доступа, задаваемую спецификацией J2SE. В Oracle 8.1.5 для этого приходилось использовать роли. К сожалению, роль была ровно одна: **JAVASYSPRIV.** Ее использование будет подобно предоставлению роли администратора базы данных каждому пользователю только потому, что ему необходимо создать представление, — это слишком мощная роль для выполнения такого простого действия. При наличии роли **JAVASYSPRIV** можно делать все, что угодно. Будьте осторожны при использовании этой роли в версии 8.1.5 и постарайтесь перейти на следующие версии, где принята более избирательная модель привилегий.

Выполнение команды ОС

Если бы я получал десятицентовую монету всякий раз, отвечая на вопрос о том, как выполнить команду ОС! До появления поддержки языка Java в СУБД, это действитель-

но было сложно. Теперь же это почти тривиально. Есть, вероятно, сотни способов сделать это, но следующий фрагмент кода работает вполне удовлетворительно:

```
tkyte@TKYTE816> create or replace and compile
  2  java source named "Util"
  3  as
  4  import java.io.*;
  5  import java.lang.*;
  6
  7  public class Util extends Object
  8  {
  9
 10    public static int RunThis(String[] args)
 11    {
 12    Runtime rt = Runtime.getRuntime();
 13    int       rc = -1;
 14
 15    try
 16    {
 17       Process p = rt.exec(args[0]);
 18
 19       int bufSize = 4096;
 20    BufferedInputStream    bis =
 21       new BufferedInputStream(p.getInputStream(), bufSize);
 22      int len;
 23      byte buffer[] = new byte[bufSize];
 24
 25      // Выдаем то, что получено программой
 26     while ((len = bis.read(buffer, 0, bufSize)) != -1)
 27        System.out.write(buffer, 0, len);
 28
 29       rc = p.waitFor();
 30    }
 31    catch (Exception e)
 32    {
 33       e.printStackTrace();
 34      rc = -1;
 35    }
 36    finally
 37    {
 38       return rc;
 39    }
 40    }
 41  }
 42  /

Java created.
```

Он позволяет выполнить любую программу и получить ее результаты либо в файле трассировки на сервере, либо, при использовании средств пакета DBMS_JAVA, в буфере пакета **DBMS_OUTPUT**. Это, однако, весьма мощное средство — при наличии соответствующих привилегий с его помощью можно выполнять любую команду от име-

ни пользователя — владельца ПО Oracle. В данном случае я хочу иметь возможность получить список процессов с помощью утилиты **/usr/bin/ps** в ОС UNIX и **\bin\tlist.exe** в Windows. Для этого мне необходимы две привилегии:

```
tkyte@TKYTE816> BEGIN
  2      dbms_java.grant_permission
  3      (USER,
  4       'java.io.FilePermission',
  5       — '/usr/bin/ps',   -- для UNIX
  6    c:\bin\tlist.exe',     -- для WINDOWS
  7       'execute');
  8
  9      dbmsiava.grant_permission
 10     (USER;
 11       'java.lang.RuntimePermission',
 12      '*',
 13      'writeFileDescriptor');
 14  end;
 15  /

PL/SQL procedure successfully completed.
```

*В вашей системе может отсутствовать утилита **tlist.exe**. Она входит в состав набора инструментальных средств Windows Resource Toolkit и доступна не во всех Windom-системах. Этот пример просто показывает, что можно сделать, — отсутствие доступа к **tlist.exe** не помешает демонстрации. Этот метод можно использовать дм выполнения любой программы. Учтите, однако, что нужно быть внимательным, предоставляя права на выполнение программ с помощью пакета **DBMS_JAVA.** Например, предоставление права на выполнение программы с: \winnt \system32 \cmd.exe фактически означает разрешение выполнять ВСЕ программы, что очень опасно.*

Первый вызов **dbms_java.grant_permission** позволяет запускать одну конкретную программу. При желании можно рискнуть и указать вместо имени программы символ *. Это позволит выполнять любые программы. Я не думаю, однако, что это разумно; явно перечисляйте полные имена программ, в надежности которых вы уверены. Вторая привилегия позволяет генерировать результаты во время выполнения. Здесь придется использовать метасимвол *, поскольку я не знаю точно, куда именно будут выдаваться результаты (в стандартный выходной поток, **stdout,** например, или куда-нибудь еще).

Теперь необходимо создать уровень связывания:

```
tkyte@TKYTE816> create or replace
  2  function RUN_CMD(p_cmd  in varchar2)  return number
  3  as
  4  language java
  5  name 'Util.RunThis(java.lang.String[]) return integer';
  6  /

Function created.

tkyte@TKYTE816> create or replace procedure rc(
  2  as
```

```
  3    x number;
  4  begin
  5    x := run_cmd(p_cmd);
  6     if (x o 0)
  7    then
  8          raise program_error;
  9    end if;
 10  end;
 11  /

Procedure created.
```

Здесь я создал еще один уровень абстракции выше функции связывания, чтобы можно было выполнять программу как процедуру. Давайте посмотрим, как это работает:

```
tkyte@TKYTE816> set serveroutput on size 1000000
tkyte@TKYTE816> exec dbms_java.set_output(1000000)

PL/SQL procedure successfully completed.

tkyte@TKYTE816> exec rc('C:\WINNT\system32\cmd.exe /c dir')
Volume in drive C has no label.
Volume Serial Number is F455-B3C3
Directory of C:\oracle\DATABASE
05/07/2001  10:13a      <DIR>
05/07/2001  10:13a      <DIR>
11/04/2000  06:28p      <DIR>          ARCHIVE
11/04/2000  06:37p                    47 inittkyte816.ora
11/04/2000  06:28p                31,744 ORADBA.EXE
05/07/2001  09:07p                 1,581 oradim.log
05/10/2001  07:47p                 2,560 pwdtkyte816.ora
05/06/2001  08:43p                 3,584 pwdtkyte816.ora.hold
01/26/2001  11:31a                 3,584 pwdtkyte816.xxx
04/19/2001  09:34a                21,309 sqlnet.log
05/07/2001  10:13a                 2,424 test.sql
01/30/2001  02:10p               348,444 xml.tar
9 File(s)        415,277 bytes
3 Dir(s)  13,600,501,760 bytes free

PL/SQL procedure successfully completed.
```

Мы получили список файлов каталога ОС.

Получение времени с точностью до миллисекунд

Примеры становятся все меньше, короче и выполняются быстрее. Это я и хочу подчеркнуть. С помощью небольших фрагментов Java-кода, примененных в соответствующих местах, можно существенно расширить функциональные возможности.

В Oracle 9i эта функция станет избыточной, поскольку эта версия поддерживает временные отметки с точностью менее секунды. Но при необходимости такая точность измерения времени достижима и в предыдущих версиях:

```
tkyte@TKYTE816> create or replace java source
  2  named "MyTimestamp"
  3  as
```

```
  4 import java.lang.String;
  5 import java.sql.Timestamp;
  6
  7 public class MyTimestamp
  8  {
  9     public static String getTimestamp()
 10      {
 11        return  (new
 12         Timestamp(System.currentTimeMillis())).toString();
 13      }
 14  };
 15 /

Java created.

tkyte@TKYTE816> create or replace function my_timestamp return varchar2
  2  as language java
  3  name 'MyTimestamp.getTimestamp() return java.lang.String';
  4  /

Function created.

tkyte@TKYTE816> select my_timestamp,
  2  to_char(sysdate,'yyyy-mm-dd hh24:mi:ss') from dual
  3  /

MY)TIMESTAMP             TO_CHAR(SYSDATE,'YY

2001-03-27 19:15:59.688  2001-03-27 19:15:59
```

Возможные ошибки

Большинство сообщений об ошибках, которые вы получите при использовании хранимых процедур на Java, связаны с компиляцией кода и несоответствием типов параметров. Некоторые из наиболее типичных сообщений об ошибках рассмотрены ниже.

ORA-29549 Java Session State Cleared

По ходу разработки можно столкнуться с сообщениями следующего вида:

```
select my_timestamp,  to_char(sysdate,'yyyy-mm-dd hh24:mi:ss')  from dual
                                                                      *
ERROR at line 1:
ORA-29549: class TKYTE.MyTimestamp has changed, Java session state cleared
```

Это означает, что использованный в сеансе класс был перекомпилирован (скорм всего — вами же). Вся связанная с этим классом информация о состоянии потеряна. Достаточно повторно выполнить оператор, при выполнении которого было выдано это сообщение, и информация о состоянии обновится.

По этой причине следует избегать повторной загрузки Java-классов в действующе* производственной системе. После этого использующий Java-класс сеанс при обращении к нему получит такое сообщение об ошибке.

Ошибки прав доступа

Мы уже знакомы с таким сообщением:

```
ERROR at line 1:
ORA-29532: Java call terminated by uncaught Java exception:
java.security.AccessControlException:
the Permission (java.io.FilePermission c:\temp read) has not been granted by
dbms_java.grant_permission to
SchemaProtectionDomain(TKYTE IPolicyTableProxy(TKYTE))
ORA-06512: at "TKYTE.GET_DIR_LIST", line 0
ORA-06512: at line 1
```

К счастью, в тексте сообщения об ошибке явно указано, какие привилегии необходимо получить, чтобы вызов был успешным. Обладающий соответствующими привилегиями пользователь должен предоставить вам эти привилегии с помощью процедуры GRANT_PERMISSION пакета DBMS_JAVA.

ORA-29531 no method X in class Y

Если в рассмотренном ранее примере RunThis изменить спецификацию вызова следующим образом:

```
tkyte@TKYTE816>create or replace
  2 function RUN_CMD(p_cmd in varchar2) return number
  3 as
  4 language java
  5 name 'Util.RunThis(String[]) return integer';
  7 /

Function created.
```

будет выдано сообщение об ошибке ORA-29531. Обратите внимание, что в списке параметров функции Util.RunThis, я указал тип данных String, а не java.lang.String.

```
tkyte@TKYTE816> exec rc('c:\winnt\system32\cmd.exe /c dir')
java.lang.NullPointerException
atoracle.aurora.util.JRIExtensions.getMaximallySpecificMethod(JRIExtensions.java)
at oracle.aurora.util.JRIExtensions.getMaximallySpecificMethod(JRIExtensions.java)
BEGIN RC('c:\winnt\system32\cmd.exe /c dir'); END;
*

ERROR at line 1:
ORA-29531: no method RunThis in class Util
ORA-06512: at "TKYTE.RUN_CMD", line 0
ORA-06512: at "TKYTE.RC", line 5
ORA-06512: at line 1
```

Дело в том, что для успешного сопоставления типов данных должны указываться **полные (fully qualified)** имена типов. Хотя класс **java.lang** неявно импортируется в Java-программах, он не импортируется на уровне языка SQL. Получив это сообщение об ошибке, необходимо проверить сопоставление типов данных и убедиться, что используются **полные** имена типов данных Java и что они в точности совпадают с именами

имеющихся типов данных. Соответствующий Java-метод определяется по сигнатуре, а сигнатура создается на основе используемых типов данных. Минимальное различие в типах входных данных, результатов или регистре символов в имени приведет к несовпадению сигнатур, и сервер Oracle не найдет соответствующий код.

Резюме

В этой главе вы узнали, как реализовать хранимые процедуры на языке Java. Это не означает, что весь существующий код на языке PL/SQL необходимо переписать в виде хранимых процедур на Java. Но если, программируя на PL/SQL, вы столкнетесь с неразрешимой проблемой, что обычно происходит при необходимости выйти за пределы базы данных и обеспечить взаимодействие с операционной системой, попробуйте решить задачу с помощью языка Java.

Благодаря полученным в этой главе сведениям вы сможете передать основные типы данных SQL, в том числе массивы, с уровня PL/SQL на уровень Java-методов и получить результаты. Я представил несколько полезных фрагментов Java-кода, которые можно использовать непосредственно; обратившись к документации по языку Java, вы обнаружите десятки других фрагментов, незаменимых при разработке приложений.

При осторожном использовании, программирование на Java может существенно расширить возможности разработки приложений.

20

Использование объектно-реляционных средств

Начиная с версии Oracle 8, в базах данных сервера Oracle могут использоваться *объектно-реляционные средства*. Выходя за пределы стандартных скалярных типов NUMBER, DATE и строк символов, объектно-реляционные средства Oracle позволяют расширить набор поддерживаемых типов данных. Можно создавать собственные типы данных, включающие:

- атрибуты, каждый из которых может быть скалярной величиной или набором (массивом) других объектных/скалярных типов;
- методы для работы с данными этого типа;
- статические методы;
- необязательный метод сравнения, используемый для сортировки и сравнения данных.

Затем этот новый тип можно использовать для создания таблиц, столбцов таблиц, представлений или для расширения возможностей языков SQL и PL/SQL. Вновь созданный пользовательский тип данных можно использовать точно так же, как и базовый тип данных **DATE**.

В этой главе я продемонстрирую, как использовать объектно-реляционные средства сервера Oracle. Будет также показано, как их не следует использовать. Я буду описывать компоненты этой технологии последовательно. Однако эту главу нельзя считать полным обзором всех возможностей объектно-реляционных средств Oracle. Этому посвящено 200-страничное руководство Oracle *Application Developer's Guide—Object-Relational Features*. Цель данной главы — показать, когда и как использовать эти возможности.

Объектно-реляционные средства Oracle можно использовать во многих языках. При программировании на Java — через интерфейс JDBC, на Visual Basic — с помощью компонентов 0040 (Oracle Objects for Ole). При программировании с помощью библиотеки OCI (Oracle Call interface), языка PL/SQL и прекомпилятора Pro*C очень легко использовать соответствующие функциональные возможности. Корпорация Oracle предоставляет различные инструментальные средства, упрощающие использование объектно-реляционных возможностей сервера в этих языках. Например, при программировании на Java/JDBC можно использовать Oracle JPublisher — утилиту, автоматически генерирующую Java-классы, представляющие объектные типы базы данных, наборы и PL/SQL-пакеты (это генератор кода, который позволяет сопоставлять сложные типы SQL типам языка Java). Библиотека OCI поддерживает на стороне клиента встроенный кэш объектов, используемый для эффективного управления и обработки объектов. Прекомпилятор Pro*C включает средство OTT (Object Type Translator — транслятор объектных типов) для генерации структур **(struct)** языка C/C++, соответствующих объектным типам. Я не буду касаться использования этих языков и средств — все это детально описано в документации сервера Oracle. Все внимание будет сосредоточено на создании объектных типов в базе данных.

В каких случаях используются объектно-реляционные средства

Я использую объектно-реляционные средства сервера Oracle преимущественно для естественного расширения возможностей языка PL/SQL. Объектный тип — прекрасный способ добавить в PL/SQL новые функциональные возможности аналогично тому, как классы позволяют сделать это в C++ или Java. В следующем разделе мы рассмотрим соответствующий пример.

Объектные типы можно также использовать для стандартизации. Я могу создать новый тип, скажем, **ADDRESS_TYPE,** который инкапсулирует определение адреса или отдельных компонентов, из которых состоит адрес. Можно даже добавить служебные функции (методы) для этого типа, которые, например, возвращают адрес в формате, подходящем для распечатки на почтовых наклейках. Теперь при создании таблицы, в которой должны содержаться данные об адресе, можно просто указать столбец типа **ADDRESS_TYPE.** Атрибуты адреса при этом будут добавлены в таблицу автоматически. Пример такого использования тоже будет рассмотрен.

Объектные типы можно использовать для объектно-реляционного представления чисто реляционных, по сути, данных. Т.е. можно взять пару таблиц **EMP/DEPT** и построить объектно-реляционное их представление, в котором каждая строка таблицы **DEPT** будет содержать набор объектов **EMP.** Не соединяя таблицы **EMP** и **DEPT,** я смогу обратиться к объектному представлению **DEPT** и получить информацию из таблиц DEPT и **EMP** в одной строке. В следующем разделе мы рассмотрим и этот пример.

Объектные типы можно также использовать для создания объектных таблиц. Преимущества и недостатки объектных таблиц рассматривались в главе 6. Объектные таблицы содержат множество скрытых столбцов; при использовании этих таблиц возникают побочные эффекты, и происходят различные "чудеса". Кроме того, обычно (для

множества различных целей) необходимо строго реляционное представление данных (в частности, для утилит и средств создания отчетов, которые "не понимают" объектные типы). Именно поэтому объектные таблицы я стараюсь не использовать. Я использую объектные представления реляционных данных, что в конечном итоге дает те же возможности, что и объектные таблицы. Однако при этом я контролирую все аспекты физического хранения данных. Поэтому тему объектных таблиц я здесь подробно рассматривать не буду.

Как работают объектно-реляционные средства

В этом разделе мы рассмотрим использование объектно-реляционных средств Oracle для решения следующих задач:

- расширение набора стандартных типов данных в системе;
- естественное расширение возможностей языка PL/SQL;
- создание объектно-реляционных представлений реляционных, по сути, данных.

Добавление новых типов данных в систему

Начнем с простого: типа данных **ADDRESS_TYPE.** Рассмотрим синтаксис соответствующих конструкций, их возможности, побочные эффекты, с которыми можно столкнуться, и т.п. Для начала создадим простой тип:

```
tkyte@TKYTE816>create or replace type Address_Type
  2  as object
  3   (street_addr1   varchar2(25),
  4    street_addr2   varchar2(25),
  5    city           varchar2(30),
  6    state          varchar2(2),
  7    zip_code       number
  8   )
  9   /

Type created.
```

Это простейшая разновидность оператора **CREATE TYPE**. По ходу работы над примером мы добавим в него дополнительные конструкции. Этот тип состоит только из заданных скалярных типов, не имеет методов, специфических функций сравнения — ничего "выдающегося". Зато его можно сразу же использовать в таблицах и в PL/SQL-коде:

```
tkyte@TKYTE816>create table people
  2   (name           varchar2(10),
  3   home_address    address_type,
  4   work_address    address_type
  5   )
```

```
  6  /

Table created.

tkyte@TKYTE816> declare
  2      l_home_address address_type;
  3      l_work_address address_type;
  4  begin
  5      l_home_address := Address_Type('123 Main Street', null,
  6                        'Reston', 'VA', 45678);
  7      l_work_address := Address_Type('1 OracleWay', null,
  8                        'Redwood', 'CA', 23456);
  9
 10      insert into people
 11      (name, home_address, work_address)
 12      values
 13      ('TomKyte', l_home_address, l_work_address);
 14  end;
 15  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select * from people;

NAME     HOME_ADDRESS(STREET_WORK_ADDRESS(STREET_

Tom Kyte ADDFESS_TYPE('123 Ma ADDRESS_TYPE('1 Orac
           in Street', NULL,  'R le Way', NULL,  'Redw
           eston', 'VA', 45678) ood', 'CA', 23456)
```

Как видите, использовать этот новый тип в операторе **CREATE TABLE** так же легко, как, например, тип **NUMBER.** Кроме того, объявлять переменные типа **ADDRESS_TYPE** в PL/SQL тоже просто: в языке PL/SQL новые типы данных можно использовать сразу же. Новые функциональные возможности использованы в приведенном PL/SQL-коде в строках с 5 по 8. Здесь вызывается конструктор объекта нового типа. Стандартный конструктор типа позволяет задать значения для всех атрибутов объектного типа. По умолчанию создается только один стандартный конструктор, при вызове которого надо указать значения для всех атрибутов типа. В разделе "Использование типов для расширения возможностей PL/SQL" мы рассмотрим, как создавать специфические конструкторы с помощью статических функций-членов.

Созданные переменные типа **ADDRESS_TYPE** после инициализации можно использовать в качестве связываемых переменных в SQL-операторах, как было показано выше. Достаточно просто вставить значения столбцов **NAME, HOME_ADDRESS** и **WORK_ADDRESS.** Несложный SQL-запрос позволяет получить эти данные. С помощью SQL можно обращаться не только к столбцу **HOME_ADDRESS,** но и к каждому из компонентов **HOME_ADDRESS**. Например:

```
tkyte@TKYTE816> select name, home_address.state, work_address.state
  2    from people
  3  /
select name, home_address.state, work_address.state

ERROR at line 1:
ORA-00904: invalid column name
```

```
tkyte@TKYTE816> select name, P.home_address.state, P.work_address.state
  2    from people P
  3  /

NAME       HOME_ADDRESS.STATE   WORK_ADDRESS.STATE

Tom Kyte   VA                   CA
```

Я продемонстрировал неправильный и правильный способ. Первый пример — это то, что обычно пишут разработчики. Запрос, конечно, не работает. Чтобы обратиться к компонентам объектного типа, необходимо использовать корреляционное имя, как сделано во втором запросе. В нем я задал для таблицы **PEOPLE** псевдоним P (можно использовать любой допустимый идентификатор, включая слово **PEOPLE).** Если возникает необходимость сослаться на отдельные компоненты адреса, я использую псевдоним.

Как же выглядит в действительности таблица **PEOPLE?** То, что показывает сервер Oracle, весьма отличается от того, что используется **на самом** деле, как можно догадаться, прочитав главу 6 и изучив примеры с вложенной или объектной таблицей:

```
tkyte@TKYTE816> desc people
 Name                                      Null?    Type

 NAME                                               VARCHAR2(10)
 HOME_ADDRESS                                       ADDRESS_TYPE
 WORK_ADDRESS                                       ADDRESS_TYPE
tkyte@TKYTE816> select name, length
  2    from sys.col$
  3   where obj# = (select object_id
  4                   from user_objects
  5                  where object_name = 'PEOPLE')
  6  /

NAME                           LENGTH

NAME                               10
HOME_ADDRESS                        1
SYS_NC00003$                       25
SYS_NC00004$                       25
SYS_NC00005$                       30
SYS_NC00006$                        2
SYS_NC00007$                       22
WORK_ADDRESS                        1
SYS_NC00009$                       25
SYS_NC00010$                       25
SYS_NC00011$                       30
SYS_NC00012$                        2
SYS_NC00013$                       22

13 rows selected.
```

Сервер Oracle сообщает, что в таблице — три столбца, но в реальном словаре данных их, однако, — тринадцать. В нем можно обнаружить скрытые скалярные столбцы. Хотя все не так очевидно и используются скрытые столбцы, применять скалярные объектные типы (без вложенных таблиц) подобным образом несложно. С такого рода неоче-

видностью вполне можно смириться. Если использовать опцию SET DESCRIBE утилиты SQL*Plus, можно заставить эту утилиту показывать всю структуру объектного типа:

```
tkyte@TKYTE816> set describe depth all
tkyte@TKYTE816> desc people
 Name                                      Null?    Type

 NAME                                               VARCHAR2(10)
 HOME_ADDRESS                                       ADDRESS_TYPE
   STREET_ADDR1                                     VARCHAR2(25)
   STREET_ADDR2                                     VARCHAR2(25)
   CITY                                             VARCHAR2(30)
   STATE                                            VARCHAR2(2)
   ZIP_CODE                                         NUMBER
 WORK_ADDRESS                                       ADDRESS_TYPE
   STREET_ADDR1                                     VARCHAR2(25)
   STREET_ADDR2                                     VARCHAR2(25)
   CITY                                             VARCHAR2(30)
   STATE                                            VARCHAR2(2)
   ZIP_CODE                                         NUMBER
```

Это очень удобно для определения поддерживаемых атрибутов.

Теперь давайте немного усложним тип ADDRESS_TYPE: добавим функцию выдачи адреса в удобном формате в виде одного поля. Для этого можно добавить в тело типа соответствующую функцию-член:

```
tkyte@TKYTE816> alter type Address_Type
  2  BEPLACE
  3  as object
  4   (street_addrl   varchar2(25),
  5    street_addr2   varchar2(25),
  6    city           varchar2(30),
  7    state          varchar2(2),
  8    zip_code       number,
  9    member function toString return varchar2
 10  )
 11  /

Type altered.

tkyte@TKYTE816> create or replace type body Address_Type
  2  as
  3      member function toString return varchar2
  4      is
  5      begin
  6          if (street_addr2 is not NULL)
  7          then
  8             return street_addrl || chr(10) ||
  9                    street_addr2 || chr(10) ||
 10                 city || ', ' || state || ' ' || zip_code;
 11          else
 12             return street_addrl || chr(10) ||
 13                  city || ', ' || state || ' ' || zip_code;
```

```
14            end if;
15        end;
16    end;
17    /

Type body created.

tkyte@TKYTE816> select name, p.home_address.toString()
  2    from people P
  3    /

NAME

P.HOME_ADDRESS.TOSTRING()

Tom Kyte
123 Main Street
Reston, VA 45678
```

Вот и первый пример метода объекта. Каждый метод вызывается с неявным параметром **SELF**. Можно добавить этот префикс к атрибутам **STREET_ADDR1**, **STREET_ADDR2** и т.д.:

```
SELF.street_addr1 || chr(10) || SELF.street_addr2 ...
```

но он и так добавляется неявно. Тут вы вполне резонно можете заметить: "Ведь все это можно сделать с помощью реляционной таблицы и PL/SQL-пакета". Это действительно так. Однако использование объектного типа с методами, как показано выше, дает определенные преимущества.

- **Обеспечивается более совершенный механизм инкапсуляции.** Тип **ADDRESS_TYPE** инкапсулирует и поддерживает адрес, со всеми его атрибутами и функциональными возможностями.
- **Методы более тесно привязываются к специфическим данным.** Это очень важный момент. Если используются скалярные столбцы и PL/SQL-функция, форматирующая их для вывода адреса на печать, эту функцию можно вызвать с любыми данными. Можно передать значение столбца **EMPLOYEE_NUMBER** в качестве почтового индекса, фамилию — вместо названия улицы и т.д. Привязывая метод к атрибутам, мы гарантируем, что метод **TOSTRING** может работать только с данными адреса. Пользователи, вызывающие этот метод, не должны задумываться о передаче соответствующих данных — они "уже здесь".

Однако объектный тип имеет один недостаток: в Oracle 8i он мало поддается изменениям. С помощью оператора **ALTER** можно добавлять новые методы, но нельзя ни **удалить** существующие, ни **добавить** дополнительные атрибуты после создания таблицы, использующей этот тип (да и удалить добавленные методы невозможно). Единственное, что можно делать, — это добавлять методы или изменять их (тело типа). Другими словами, развитие схемы при использовании объектных типов затруднено. Если со временем окажется, что для объекта **ADDRESS_TYPE** необходим еще один атрибут, придется пересоздавать объекты, в которые этот тип встроен. Это не относится к объектным типам, которые не используются для столбцов таблиц базы данных или в операторах

CREATE TABLE OF TYPE. Другими словами, если объектные типы используются исключительно в объектных представлениях или для расширения возможностей PL/SQL (как описано в следующих разделах), этой проблемой можно пренебречь.

С объектными типами также связаны специфические методы **MAP** и **ORDER.** Они используются при сортировке, сравнении или группировке экземпляров объектных типов. Если у объектного типа нет функции **MAP** или **ORDER,** при попытке выполнения этих операций вы получите следующее сообщение об ошибке:

```
tkyte@TKYTE816>select * from people order by home_address;
select * from people order by home_address
       *
ERROR at line 1:
ORA-22950: cannot ORDER objects without MAP or ORDER method

tkyte@TKYTE816>select * from people where home_address > work_address;
select * from people where home_address > work_address
ERROR at line 1:
ORA-22950: cannot ORDER objects without MAP or ORDER method

tkyte@TKYTE816>select * from people where home_address = work_address;
no rows selected
```

Упорядочивать данные объектного типа, использовать их при поиске "больших" или "меньших" значений будет невозможно. Значения можно будет сравнивать только на равенство. При этом сервер Oracle выполняет сравнение по всем атрибутам, чтобы узнать, не совпадают ли они. Чтобы можно было выполнять все остальные операции, необходимо добавить метод **MAP** или метод **ORDER** (объектный тип может иметь либо метод **MAP**, либо метод **ORDER**, но не оба одновременно).

Метод **MAP** — это функция, работающая с одним экземпляром объекта и возвращающая значение одного из скалярных типов, которое сервер Oracle будет использовать для сравнения с другими однотипными объектами. Например, если объектный тип представляет точку на плоскости с координатами **X** и **Y**, функция **MAP** может возвращать квадратный корень из (**X*X+Y*Y**) — расстояние от начала координат. Метод **ORDER** принимает два экземпляра объекта — **SELF** и объект для сравнения с **SELF**. Метод **ORDER** возвращает 1, если **SELF** больше этого объекта, -1, если SELF меньше другого объекта или 0, если объекты равны. Метод **MAP** предпочтительнее, поскольку работает намного быстрее и даже может вызываться в параллельных запросах (метод **ORDER** нельзя использовать при распараллеливании). Метод **MAP** достаточно вызвать для экземпляра объекта один раз, и после этого сервер Oracle может использовать это значение при сортировке. Метод **ORDER** при сортировке большого множества, возможно, придется вызывать сотни или тысячи раз с одними и теми же данными. Для созданного ранее типа **ADDRESS_TYPE** я продемонстрирую оба метода. Сначала — метод ORDER

```
tkyte@TKYTE816>alter type Address_Type
  2  REPLACE
  3  as object
  4   (street_addr1   varchar2(25),
  5    street_addr2   varchar2(25),
```

```
  6   city           varchar2(30),
  7   state          varchar2(2),
  8   zip_code       number,
  9   member function toString return varchar2,
 10   order member function order_function(compare2 in Address_type)
 11   return number
 12  )
 13  /
```

Type altered.

```
tkyte@TKYTE816>  create or replace type body Address_Type
  2  as
  3      member function toString return varchar2
  4      is
  5      begin
  6          if (street_addr2 is not NULL)
  7          then
  8             return street_addrl || chr(10) ||
  9                    street_addr2 || chr(10) ||
 10                  city || ', ' || state || ' ' || zip_code;
 11          else
 12             return street_addrl || chr(10) ||
 13                  city || ', ' || state || ' ' || zip_code;
 14         end if;
 15      end;
 16
 17      order member function order_function(compare2 in Address_type)
 18      return number
 19      is
 20      begin
 21          if (nvl(self.zip_code,-99999) <> nvl(compare2.zip_code,-99999))
 22          then
 23              return sign(nvl(self.zip_code,-99999)
 24                          - nvl(compare2.zip_code,-99999));
 25         end if;
 26          if (nvl(self.city,chr(0)) > nvl(compare2.city,chr(0)))
 27          then
 28              return 1;
 29          elsif (nvl(self.city,chr(0)) <nvl(compare2.city,chr(0)))
 30          then
 31             return -1;
 32         end if;
 33          if (nvl(self.street_addrl,chr(0)) >
 34                      nvl(compare2.street_addr1,chr(0)))
 35          then
 36              return 1;
 37          elsif (nvl(self.street_addrl,chr(O)) <
 38                      nvl(compare2.street_addr1,chr(0)))
 39          then
 40              return -1;
 41         end if;
 42          if (nvl(self.street addr2,chr(0)) >
```

```
43                          nvl(compare2.street_addr2, chr(0)))
44          then
45              return 1;
46         elsif (nvl(self.street_addr2,chr(0)) <
47                     nvl(compare2.street_addr2,chr(0)))
48          then
49             return -1;
50          end if;
51          return 0;
52      end;
53  end;
54  /

Type body created.
```

Этот метод сравнивает два адреса по следующему алгоритму.

1. Если значение почтового индекса (ZIP_CODE) у объекта SELF меньше, чем у объекта **COMPARE2**, вернуть -1, а если больше — 1.
2. Если значение города (**CITY**) у объекта **SELF** меньше, чем у объекта **COMPARE2**, вернуть -1, а если больше — 1.
3. Если значение первого компонента адреса (**STREET_ADDR1**) у объекта **SELF** меньше, чем у объекта **COMPARE2**, вернуть -1, а если больше — 1.
4. Если значение второго компонента адреса (STREET_ADDR2) у объекта SELF меньше, чем у объекта **COMPARE2**, вернуть -1, а если больше — 1.
5. Иначе вернуть 0 (адреса совпадают).

Как видите, при сравнении приходится постоянно проверять, не переданы ли значения **NULL**, и т.п. В результате метод получился достаточно большим и сложным. Он, несомненно, неэффективен. Задумав написать метод ORDER, попробуйте использовать вместо него метод **MAP**. Представленное выше сравнение лучше переписать в виде метода **MAP**. Учтите, что если вы уже изменили тип, добавив в него представленный выше метод **ORDER**, придется удалить таблицу, зависящую от этого типа, удалить сам тип и создать все заново. Методы нельзя удалять — их можно только добавлять с помощью оператора **ALTER TYPE**, а нам надо избавиться от существующего метода **ORDER**. Полный пример должен был бы включать операторы **DROP TABLE PEOPLE**, **DROP TYPE ADDRESS_TYPE** и **CREATE TYPE** и лишь затем — следующий оператор **ALTER** TYPE:

```
tkyte@TKYTE816>alter type Address_Type
  2  REPLACE
  3  as object
  4   (street_addr1   varchar2(25),
  5    street_addr2   varchar2(25),
  6    city           varchar2(30),
  7    state          varchar2(2),
  8    zip_code       number,
  9   member function toString return varchar2,
 10   map member function mapping_function return varchar2
```

```
 11 )
 12 /

Type altered.

tkyte@TKYTE816>create or replace type body Address_Type
  2 as
  3    member function toString return varchar2
  4     is
  5     begin
  6        if (street_addr2 is not NULL)
  7         then
  8           return street_addrl || chr(10) ||
  9                  street_addr2 || chr(10) ||
 10                city || ', ' || state || ' ' || zip_code;
 11         else
 12    returnstreet_addr1     ||     chr(10) ||
 13                city || ', ' || state || ' ' || zip_code;
 14       end if;
 15    end;
 16
 17    map member function mapping_function return varchar2
 18     is
 19     begin
 20       returnto_char(nvl(zip_code,0), 'fm00000') ||
 21            lpad(nvl(city,' ') , 30) ||
 22       lpad(nvl(street_addr1,'       ') , 25) ||
 23       lpad(nvl(street_addr2,'       ') , 25);
 24    end;
 25 end;
 26 /
Type body created.
```

Возвращая строку фиксированной длины, содержащую значение **ZIP_CODE,** затем — CITY и поля **STREET_ADDR,** можно переложить задачу сравнений и сортировки на сервер Oracle.

Прежде чем переходить к другим вариантам использования объектных типов (я больше всего люблю использовать их для расширения возможностей языка PL/SQL), хочу представить еще один тип наборов — **VARRAY. В** главе 6 мы рассматривали вложенные таблицы и их реализацию. Было показано, что они реализуются в виде пары родительской и дочерней таблиц, со скрытым суррогатным ключом в родительской таблице и столбцом **NESTED_TABLE_ID** в порожденной. Массив **VARRAY** во многом похож на вложенную таблицу, но реализуется абсолютно иначе.

Массив **VARRAY** (или вложенная таблица) используется для хранения массива данных, связанных с одной строкой. Например, если необходимо хранить в таблице **PEOPLE** дополнительные таблицы (скажем, массив прежних адресов проживания, начиная с самого старого), можно сделать следующее:

```
tkyte@TKYTE816>create or replace type Address_Array_Type
  2 as varray(25) of Address_Type
  3 /
```

```
Type created.

tkyte@TKYTE816> alter table people add previous_addresses
Address_Array_Type
  2  /

Table altered.

tkyte@TKYTE816> set describe depth all
tkyte@TKYTE816> desc people
 Name                                      Null?    Type

 NAME                                               VARCHAR2(10)
 HOME_ADDRESS                                       ADDRESS_TYPE
   STREET_ADDR1                                     VARCHAR2(25)
   STREET_ADDR2                                     VARCHAR2(25)
   CITY                                             VARCHAR2(30)
   STATE                                            VARCHAR2(2)
   ZIP_CODE                                         NUMBER
METHOD

 MEMBER FUNCTION TOSTRING RETURNS VARCHAR2
METHOD

 MAP MEMBER FUNCTION MAPPING_FUNCTION RETURNS VARCHAR2
 WORK_ADDRESS                                       ADDRESS_TYPE
   STREET_ADDR1                                     VARCHAR2(25)
   STREET_ADDR2                                     VARCHAR2(25)
   CITY                                             VARCHAR2(30)
   STATE                                            VARCHAR2(2)
   ZIP_CODE                                         NUMBER
METHOD

 MEMBER FUNCTION TOSTRING RETURNS VARCHAR2
METHOD

 MAP MEMBER FUNCTION MAPPING_FUNCTION RETURNS VARCHAR2
 EREVIOUS_ADDRESSES                                ADDRESS_ARRAY_TYPE
   STREET_ADDR1                                     VARCHAR2(25)
   STREET_ADDR2                                     VARCHAR2(25)
   CITY                                             VARCHAR2(30)
   STATE                                            VARCHAR2(2)
   ZIP_CODE                                         NUMBER
METHOD

 MEMBER FUNCTION TOSTRING RETURNS VARCHAR2
METHOD

 MAP MEMBER FUNCTION MAPPING_FUNCTION RETURNS VARCHAR2
```

Итак, теперь в нашей таблице можно хранить до 25 предыдущих адресов. Вопрос в том, что при этом происходит "за кадром"? Обратившись к словарю данных, можно увидеть следующее:

```
tkyte@TKYTE816> select name, length
  2    from sys.col$
  3   where obj# = (select object_id
  4                   from user_objects
  5                  where object_name = 'PEOPLE')
  6  /

NAME                      LENGTH

NAME                          10
HOME_ADDRESS                   1
SYS_NC00003$                  25
SYS_NC00004$                  25
SYS_NC00005$                  30
SYS_NC00006$                   2
SYS_NC00007$                  22
WORK_ADDRESS                   1
SYS_NC00009$                  25
SYS_NC00010$                  25
SYS_NC00011$                  30
SYS_NC00012$                   2
SYS_NC00013$                  22
PREVIOUS_ADDRESSES          2940

14 rows selected.
```

Сервер Oracle добавил столбец размером 2940 байт для реализации массива VARRAY. Данные массива VARRAY будут храниться тут же, в самой строке. При этом возникает интересный вопрос: что произойдет, если размер массива станет больше 4000 байт (это максимальный размер структурированного столбца, поддерживаемый сервером Oracle)? Если удалить столбец и пересоздать его как VARRAY(50), произойдет следующее:

```
tkyte@TKYTE816> alter table people drop column previous_addresses
  2  /

Table altered.

tkyte@TKYTE816> create or replace type Address_Array_Type
  2  as varray(50) of Address_Type
  3  /

Type created.

tkyte@TKYTE816> alter table people add previous_addresses
Address_Array_Type
  2  /

Table altered.

tkyte@TKYTE816> select object_type, object_name,
  2         decode(status,'INVALID','*','')          status,
  3                  tablespace_name
  4  from user_objects a, user_segments b
```

```
  5  where a.object_name - b.segment_name  (+)
  6  order by object_type, object_name
  7  /
```

```
OBJECT_TYPE  OBJECT_NAME                    S TABLESPACE_NAME

LOB          SYS_LOB0000026301C00014$$      DATA

TABLE        PEOPLE                         DATA

TYPE         ADDRESS_ARRAY_TYPE
             ADDRESS_TYPE

TYPE BODY    ADDRESS_TYPE
```

```
tkyte@TKYTE816> select name, length
  2    from sys.col$
  3   where obj# - (select object_id
  4                   from user_objects
  5                  where object_name =  'PEOPLE')
  6  /
```

```
NAME                      LENGTH

NAME                          10
HOME_ADDRESS                   1
SYS_NC00003$                  25
SYS_NC00004$                  25
SYS_NC00005$                  30
SYS_NC00006$                   2
SYS_NC00007$                  22
WORK_ADDRESS                   1
SYS_NC00009$                  25
SYS_NC00010$                  25
SYS_NC00011$                  30
SYS_NC00012$                   2
SYS_NC00013$                  22
PREVIOUS_ADDRESSES          3852

14 rows selected.
```

Как видите, теперь сервер Oracle автоматически создал большой объект. Если объем данных в массиве **VARRAY** не превышает примерно 4000 байтов, данные хранятся вместе со строкой (inline). Если же объем данных больше, массив **VARRAY** выносится из строки в сегмент большого объекта (как и любой большой объект).

Массивы **VARRAY** либо хранятся в строке как столбец типа **RAW**, либо (при достаточно большом объеме) как большой объект. Дополнительных ресурсов для поддержки данных типа **VARRAY** (по сравнению с вложенной таблицей) надо очень мало, что делает массив **VARRAY** привлекательным методом хранения повторяющихся данных. Поиск по массиву **VARRAY** можно реализовать, преобразовав его данные в таблицу, что сделает его не менее гибким, чем вложенные таблицы:

```
tkyte@TKYTE816> update people
  2     set previous_addresses = Address_Array_Type(
  3                         Address_Type('312 Johnston Dr', null,
```

```
  4                                   'Bethlehem', 'PA', 18017),
  5                        Address_Type('513 Zulema St', 'Apartment #3",
  6                     'Pittsburg',                  'PA', 18123),
  7                          Address_Type('840 South Frederick St', null,
  8                                   'Alexandria', 'YA', 20654));

1 row updated.

tkyte@TKYTE816> select name, prev.city, prev.state, prev.zip_code
  2    from people p, table(p.previous_addresses) prev
  3   where prev.state = 'PA';

NAME                 CITY                           ST    ZIP_CODE

Tom Kyte             Bethlehem                      PA       18017
Tom Kyte             Pittsburg                      PA       18123
```

Существенное различие состоит в том, что при реализации с помощью вложенной таблицы можно создать индекс по столбцу **STATE** вложенной таблицы, и оптимизатор этот индекс использовал бы. В данном случае столбец **STATE** проиндексировать нельзя.

Итак, основные отличия вложенных таблиц от массивов переменной длины **(VARRAY)** представлены в следующей таблице.

Вложенная таблица	***VARRAY***
Элементы "массива" не упорядочены. Данные из набора могут возвращаться совсем не в том порядке, в каком они туда вставлялись.	**VARRAY** — настоящие массивы. Данные после вставки остаются упорядоченными. В рассмотренном ранее примере данные добавлялись в конец массива. Это означает, что самый старый адрес идет в массиве первым, а последний по времени адрес находится в конце массива. При использовании внешней таблицы для упорядочения адресов по давности потребуется дополнительный атрибут.
Вложенные таблицы физически хранятся в виде пары родительской и дочерней таблиц с суррогатными ключами.	Массивы **VARRAY** хранятся как столбец типа **RAW** или как большой объект. При этом для обеспечения их работы требуются минимальные дополнительные ресурсы.
Вложенные таблицы не имеют ограничения на количество хранящихся элементов.	Для массивов **VARRAY** при создании типа задается ограничение на количество хранящихся элементов.
Вложенные таблицы можно изменять (добавлять/изменять/удалять в них данные) с помощью языка SQL.	Массивы **VARRAY** необходимо изменять процедурно. Нельзя выполнять операторы вида: **INSERT INTO** **TABLE (SELECT P.PREVIOUS ADDRESSES FROM PEOPLE P)** **VALUES ...** как для вложенной таблицы. Для добавления адреса придется использовать процедурный код (см. пример ниже).

Вложенная таблица	*VARRAY*
Для получения данных строки при использовании вложенных таблиц необходимо выполнять реляционное соединение. В случае небольших наборов это повлечет чрезмерное использование ресурсов.	Для получения данных массива VARRAY соединение выполнять не нужно. В случае небольших наборов данные хранятся в самой строке; если же наборы большие - в сегменте большого объекта. При обращении к элементам массива VARRAY требуется меньше ресурсов, чем при обращении к вложенной таблице. При изменении же данных массива VARRAY ресурсов требуется больше, чем при изменении вложенной таблицы, поскольку заменять приходится весь массив, а не один элемент.

В представленной выше таблице указано, что массив **VARRAY** нельзя изменять с помощью SQL-операторов с конструкцией **TABLE**; его надо обрабатывать процедурно. Для изменения столбцов типа **VARRAY** лучше всего написать хранимую процедуру. Ее код может быть примерно таким:

```
tkyte@TKYTE816> declare
  2    l_prev_addresses   address_Array_Type;
  3  begin
  4          select p.previous_addresses into l_prev_addresses
  5            from people p
  6           where p.name = 'Tom Kyte';
  7
  8           l_prev_addresses.extend;
  9          l_prev_addresses(l_prev_addresses.count)  :=
 10           Address_Type('123 Main Street', null,
 11                        'Reston', 'VA', 45678);
 12
 13         update people
 14            set previous_addresses = l_prev_addresses
 15           where name = 'Tom Kyte';
 16 end;
 17 /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select name, prev.city, prev.state, prev.zip_code
  2    from people p, table(p.previous_addresses) prev
  3  /

NAME                 CITY                 ST ZIP_CODE
-------------------- -------------------- -- --------
Tom Kyte             Bethlehem            PA    18017
Tom Kyte             Pittsburg            PA    18123
Tom Kyte             Alexandria           VA    20654
Tom Kyte             Reston               VA    45678
```

Мы рассмотрели преимущества и недостатки использования расширенных типов данных Oracle в таблицах базы данных. Вы должны решить, стоит ли ради возможности

создавать новые типы данных с однозначно реализованными методами обработки и использовать их в определениях столбцов, жертвовать возможностью развивать эти типы со временем (добавлять или удалять атрибуты).*

Мы также сравнили использование массивов **VARRAY** и вложенных таблиц как способов физического хранения данных. Было показано, что массивы **VARRAY** больше подходят для хранения ограниченного набора упорядоченных элементов, чем вложенные таблицы. Использовать массивы **VARRAY** очень удобно для хранения списка элементов, которому не требуется отдельная таблица.

Избирательное использование новых типов существенно улучшает систему и ее структуру. Использование объектных типов Oracle в качестве типов столбцов таблиц (но не создание объектных таблиц, как было продемонстрировано в главе 6) позволяет обеспечить стандартизацию и вызов процедур (методов) с семантически правильными параметрами. Огорчает только невозможность существенно развивать тип данных схемы после того, как создана хотя бы одна таблица, в которой этот тип используется.

Использование типов для расширения возможностей языка PL/SQL

Именно в этом объектно-реляционные средства Oracle максимально преуспели. Язык PL/SQL — очень гибкий и мощный, как доказывает уже то, что механизм расширенной репликации (Advanced Replication) был написан полностью на PL/SQL еще в версии Oracle 7.1.6. Приложения из набора Oracle Applications (Human Resources — управление персоналом, Financial Applications — бухгалтерский учет, CRM applications — управление ресурсами и т.д.) разработаны в основном на PL/SQL. Хотя это и замечательный язык программирования, встречаются ситуации, когда его базовые возможности требуется расширить (как и в случае языков Java, C, C++ или любых других языков программирования). Это можно сделать с помощью объектных типов. Они добавляют новые функциональные возможности в PL/SQL, как классы — в языках Java или C++.

В этом разделе я продемонстрирую, как использовать объектные типы для упрощения программирования на PL/SQL. Будет создан тип данных **File** на основе средств пакета **UTL_FILE. UTL_FILE** — это стандартный пакет, поставляемый в составе сервера Oracle и позволяющий выполнять в PL/SQL операции ввода-вывода (чтение и запись) текстовых данных в файлы на сервере. Он обеспечивает функциональный интерфейс, аналогичный семейству f-функций языка C **(fopen, fclose, fread, fwrite** и т.д.). Функциональные возможности пакета **UTL_FILE** будут инкапсулированы в простой в использовании объектный тип.

Создание нового типа данных PL/SQL

Пакет **UTL_FILE** возвращает записи PL/SQL (данные типа **RECORD).** Это несколько усложняет работу, но проблему можно решить. Усложнение связано с тем, что объектный тип SQL может содержать только SQL-типы, но не типы данных PL/SQL. Поэтому

'Хочу отметить, что в версии Oracle 9i ситуация принципиально меняется, поскольку появляется возможность развивать систему типов за счет наследования. - **Прим. научн. ред.**

нельзя создать объектный тип, содержащий атрибут типа записи PL/SQL, но нам это необходимо, чтобы инкапсулировать функциональные возможности существующего пакета. Чтобы решить эту проблему, придется создать вместе с типом небольшой PL/SQL-пакет.

Начнем со спецификации типа — прототипа того, что мы планируем создать:

```
tkyte@TKYTE816> create or replace type FileType
  2  as object
  3  (g_file_name    varchar2(255),
  4   g_path         varchar2(255),
  5   g_file_hcLL     number,
  6
  7   static function open(p_path        in varchar2,
  8                        p_file_name   in varchar2,
  9                        p_mode        in varchar2 default 'r',
 10             p_maxlinesize            in number default 32765)
 11   return FileType,
 12
 13   member function isOpen return boolean,
 14   member procedure close,
 15   member function get_line return varchar2,
 16   member procedure put(p_text in varchar2),
 17   member procedure new_line(p_lines in number default 1),
 18   member procedure put_line(p_text in varchar2),
 19   member procedure putf(p_fmt in varchar2,
 20                         p_arg1 in varchar2 default null,
 21                         p_arg2 in varchar2 default null,
 22                         p_arg3 in varchar2 default null,
 23                         p_arg4 in varchar2 default null,
 24                         p_arg5 in varchar2 default null),
 25   member procedure flush,
 26
 27   static procedure write_io(p_file      in  number,
 28                             p_operation in  varchar2,
 29                             p_parm1     in  varchar2 default null,
 30                             p_parm2     in  varchar2 default null,
 31                             p_parm3     in  varchar2 default null,
 32                             p_parm4     in  varchar2 default null,
 33                             p_parm5     in  varchar2 default null,
 34                             p_parm6     in  varchar2 default null)
 35  )
 36  /

Type created.
```

Эта спецификация очень похожа на спецификацию пакета **UTL_FILE** (если вы не знакомы с пакетом **UTL_FILE,** можете прочитать о нем в приложении А). Он обеспечивает практически те же функциональные возможности, что и пакет **UTL_FILE,** просто в более удобном (как мне кажется) виде. Помните, при рассмотрении создания типа **ADDRESS_TYPE** я говорил, что каждый объектный тип имеет один стандартный конструктор и в этом конструкторе надо задать значения для всех атрибутов типа. Пользо-

вательский код этот стандартный конструктор не выполняет. Другими словами, он может использоваться **только** для установки атрибутов объектного типа. Это не слишком удобно. Статическая функция **OPEN** в представленном выше типе будет использоваться для демонстрации создания собственных, куда более полезных (и сложных), конструкторов для типов. Обратите внимание, что функция **OPEN** — часть объектного типа **FILETYPE** сама возвращает данные типа **FILETYPE.** Она выполняет необходимую настройку и возвращает полностью инициализированный объект. Именно для этого в основном используются статические функции-члены в объектных типах: с их помощью создают сложные конструкторы объектов. Статические функции и процедуры в объектном типе отличаются от остальных процедур и функций тем, что не получают неявного параметра SELF. Эти функции похожи на функции или процедуры пакета. Они пригодятсядля реализации общих утилит, вызываемых другими методами, но не требующих доступа к данным экземпляра (атрибутам объекта). Процедура **WRITE_IO** в представленном выше объектном типе — пример такого рода утилиты. Я использую ее для обращения к пакету **UTL_FILE,** связанного с записью в файл, так что не приходится каждый раз повторять 14-строчный блок обработки исключительных ситуаций.

Обратите внимание, что в этом объектном типе нет ссылок на тип данных **UTL_FILE.FILE_TYPE,** поскольку атрибуты объектного типа могут быть только SQL-типов. Эту запись необходимо сохранить в другом месте. Для этого я собираюсь использовать PL/SQL-пакет следующего вида:

```
tkyte@TKYTE816>create or replace package FileType_pkg
  2  as
  3     type utl_fileArrayType is table of utl_file.file_type
  4           index by binary_integer;
  5
  6     g_files utl_fileArrayType;
  7
  8    g_invalid_path_msg constant varchar2(131) default
  9    'INVALID_PATH: Недопустимое местонахождение или имя файла.';
 10
 11    g_invalid_mode_msg constant varchar2(131) default
 12  'INVALID_MODE:  Недопустимый параметр open_mode %s в вызове FOPEN.';
 13
 14    g_invalid_filehandle_msg constant varchar2(131) default
 15    'INVALID_FILEHANDLE: Недопустимый дескриптор файла.';
 16
 17    g_invalid_operation_msg constant varchar2(131) default
 18    'INVALID_OPERATION: Файл нельзя открыть или обработать так, '||
 19     'как запрошено.';
 20
 21    g_read_error_msg constant varchar2(131) default
 22     'READ_ERROR: В ходе операции чтения произошла ошибка '||
 23     'операционной системы.';
 24
 25    g_write_error_msg constant varchar2(131) default
 26     'WRITE_ERROR: В ходе операции записи произошла ошибка '||
 27     'операционной системы.';
 28
```

```
 29    g_internal_error_msg constant varchar2(131) default
 30   'INTERNAL_ERROR:   Неопределенная ошибка в PL/SQL.';
 31
 32       g_invalid_maxlinesize_msg constant varchar2(131) default
 33       'INVALID_MAXLINESIZE: Указанный максимальный размер строки %d - 'II
 34       'слишком велик или слишком мал';
 35 end;
 36 /

Package created.
```

Этот пакет будет использоваться для хранения записей типа **UTL_FILE.FILE_TYPE** в процессе выполнения. Каждый экземпляр объектного типа (переменная) **FILE_TYPE** будет выделять себе пустой "слот" в представленном выше массиве **G_FILES.** Это показывает, как создавать "приватные" данные в объектных типах Oracle. Реальные данные времени выполнения будут храниться в переменной пакета G_FILES, а в объектном типе — только дескриптор (индекс в массиве). В текущей реализации объектов в Oracle все данные объектного типа — общедоступны. Невозможно создать скрытый атрибут типа, недоступный для пользователей. Например, в случае представленного выше типа **FILE_TYPE** вполне можно обратиться к переменной экземпляра **G_FILE_NAME** непосредственно. Если это нежелательно, необходимо скрыть эту переменную в PL/SQL-пакете так же, как мы скрыли там тип PL/SQL-записи. Никто не сможет обратиться данным в PL/SQL-пакете, не получив привилегию **EXECUTE** для этого пакета, поэтому данные защищены.

Этот пакет также используется для хранения ряда констант. Объектные типы не поддерживают неизменяемые данные, поэтому пакет представляет собой удачное место для их хранения.

Я предпочитаю называть пакет, поддерживающий тип подобным образом, так, чтобы в его имя входило имя типа. Поскольку мы создали тип **FILETYPE,** для его поддержки создан пакет **FILETYPE_PKG.** Теперь можно переходить к телу типа **FILETYPE.** Оно будет содержать все представленные ранее методы, статические функции и процедуры. Ниже приведен код с комментариями.

```
tkyte@TKYTE816>create or replace type body FileType
  2  as
  3
  4  static function open(p_path      in varchar2,
  5                       p_file_name  in varchar2,
  6                       p_mode       in varchar2 default 'r',
  7                       p_maxlinesize in number default 32765)
  8  return FileType
  9  is
 10     l_file_hdl number;
 11     l_utl_file_dir varchar2(1024);
 12  begin
 13     l_file_hdl := nvl(fileType_pkg.g_files.last, 0)+1;
 14
 15     filetype_pkg.g_files(l_file_hdl) :=
 16          utl_file.fopen(p_path, p_file_name, p_mode, p_maxlinesize);
```

```
17
18      return fileType(p_file_name, p_path, l_file_hdl);
```

Представленная выше часть статической функции **OPEN** отвечает за поиск свободного слота в приватных данных (скрытых в пакете **fileype_pkg).** Для этого она добавляет единицу к значению атрибута **LAST** PL/SQL-таблицы. Если таблица пуста, **LAST** имеет значение **NULL,** поэтому его значение передается функции NVL, и первое выделяемое значение будет иметь индекс]. Следующее — 2 и так далее. Функция **CLOSE** будет удалять записи при закрытии файла, так что место в массиве при открытии и закрытии файлов будет использоваться повторно. Остальная часть функции очень проста; она открывает указанный файл и возвращает готовый к использованию, полностью проинициализированный экземпляр объекта **FILETYPE.** Далее в функции **FILETYPE.OPEN** идет блок обработки исключительных ситуаций, позволяющий перехватить и обработать все ошибки, которые могут возникнуть при выполнении функции **UTL_FILE.FOPEN:**

```
19  exception
20      when utl_file.invalid_path then
21          begin
22              execute immediate 'select value
23                                   from v$parameter
24                                  where name = ''utl_file_dir'''
25               into l_utl_file_dir;
26          exception
27              when others then
28                  l_utl_file_dir := p_path;
29        end;
30         if (instr( l_utl_file_dir||',', p_path || ',') = 0 )
31         then
32              raise_application_error
33             (-20001,'Каталог ' || p_path ||
34               ' не входит в список каталогов utl_file_dir "' ||
35                 l_utl_file_dir || ' " ' ) ;
36          else
37              raise_application_error
38               (-20001,fileType_pkg.g_invalid_path_msg);
39       end if;
40      when utl_file.invalid_mode then
41          raise_application_error
42           (-20002,replace(fileType_pkg.g_invalid_mode_msg,'%s',p_mode));
43      when utl_file.invalid_operation then
44          raise_application_error
45           (-20003,fileType_pkg.g_invalid_operation_msg);
46      when utl_file.internal_error then
47          raise_application_error
48           (-20006,fileType_pkg.g_internal_error_msg);
49      when utl_file.invalid_maxlinesize then
50          raise_application_error
51         (-20007, replace(fileType_pkg.g_invalid_maxlinesize_msg,
52                          '%d',p_maxlinesize));
53  end;
```

Блок обработки исключительных ситуаций создан для перехвата и повторного возбуждения исключительных ситуаций **UTL_FILE** более удобным образом, чем это делает пакет **UTL_FILE.** Вместо получения в вызывающей подпрограмме обычного сообщения об ошибке **(SQLERRM) USER DEFINED EXCEPTION,** мы получим нечто более осмысленное, вроде: **Недопустимый режим открытия файла.** Кроме того, для исключительной ситуации **INVALID_PATH,** которая возбуждается в том случае, когда файл нельзя открыть из-за неверного имени файла или каталога, выполняются дополнительные проверки, и причина ошибки устанавливается более точно. Если владелец этого типа имеет привилегию **SELECT** на представление **SYS.V_$PARAMETER,** мы выбираем из него значение параметра инициализации **UTL_FILE_DIR** и проверяем, можно ли использовать тот каталог, который мы пытаемся использовать. Если нельзя, выдается соответствующее сообщение. Из всех ошибок, происходящих в процессе работы с пакетом **UTL_FILE,** эта, несомненно, самая "популярная". Выдавая такое точное сообщение об ошибке, мы сэкономим многие часы отладки для начинающих пользователей пакета **UTL_FILE.**

Продолжая рассмотрение, переходим к методу **Open:**

```
member function isOpen return boolean
 is
begin
  return utl_file.is_open(filetype_pkg.g_files(g_file_hdl));
end;
```

Это просто оболочка для существующей функции UTL_FILE.IS_OPEN. Поскольку эта функция пакета UTL_FILE никогда не возбуждает исключительных ситуаций, ее реализация очень проста. Далее идет более сложный метод GET_LINE:

```
member function get_line return varchar2
 is
     l_buffervarchar2(32765);
begin
      utl_file.get_line(filetype_pkg.g_files(g_file_hdl), l_buffer);
     return l_buffer;
exception
     when utl_file.invalid_filehandle then
        raise_application_error
       (-20002,fileType_pkg.g_invalid_filehandle_msg);
   when utl_file.invalid_operation then
       raise_application_error
       (-20003,fileType_pkg.g_invalid_operation_msg);
   when utl_file.read_error then
       raise_application_error
       (-20004,fileType_pkg.g_read_error_msg);
   when utl_file.internal_error then
       raise_application_error
       (-20006,fileType_pkg.g_internal_error_msg);
end;
```

В нем используется локальная переменная типа VARCHAR2(32765), — переменная максимально возможного в PL/SQL размера и одновременно самая длинная строка,

которую фактически позволяет прочитать пакет **UTL_FILE**. Как и в представленном ранее методе **OPEN**, мы перехватываем и обрабатываем исключительные ситуации, которые возбуждаются подпрограммой **UTL_FILE.GET_LINE**, и преобразуем их в вызовы **RAISE_APPLICATION_ERROR**. Это позволяет выдавать информативные сообщения об ошибках в функции **GET_LINE** (для удобства использования **GET_LINE** реализована как функция, а не как процедура).

Теперь переходим к другой статической процедуре — **WRITE_IO**. Процедура **WRITE_IO** используется единственно для того, чтобы избежать написания одних и тех же обработчиков исключительных ситуаций шесть раз, для каждой из подпрограмм, связанных с записью, поскольку все они возбуждают одни и те же исключительные ситуации. Эта функция, добавленная исключительно для удобства программирования, просто вызывает одну из шести функций пакета **UTL_FILE** и обрабатывает все возможные ошибки:

```
 82  static procedure write_io(p_file      in number,
 83                            p_operation in varchar2,
 84                            p_parm1     in varchar2 default null,
 85                            p_parm2     in varchar2 default null,
 86                            p_parm3     in varchar2 default null,
 87                            p_parm4     in varchar2 default null,
 88                            p_parm5     in varchar2 default null,
 89                            p_parm6     in varchar2 default null)
 90 is
 91     l_file utl_file.file_type default  filetype_pkg.g_files(p_file);
 92  begin
 93      if    (p_operation='close')    then
 94         utl_file.fclose(l_file);
 95     elsif (p_operation='put')      then
 96         utl_file.put(l_file,p_parm1) ;
 97     elsif (p_operation='new_line') then
 98         utl_file.new_line(l_file,p_parm1);
 99     elsif (p_operation='put_line') then
100          utl_file.put_line(l_file, p_parm1);
101     elsif (p_operation='flush')    then
102         utl_file.fflush(l_file);
103     elsif (p_operation='putf')     then
104         utl_file.putf(l_file,p_parm1, p_parm2,
105                       p_parm3, p_parm4, p_parm5,
106                        p_parm6);
107     else raise program_error;
108    end if;
109  exception
110      when utl_file.invalid_filehandle then
111          raise_application_error
112           (-20002,fileType_pkg.g_invalid_filehandle_msg);
113      when utl_file.invalid_operation then
114          raise_application_error
115           (-20003,fileType_pkg.g_invalid_operation_msg);
116      when utl_file.write error then
```

```
117          raise_application_error
118           (-20005,fileType_pkg.g_write_error_msg);
119      when utl_file.internal_error then
120          raise_application_error
121           (-20006,fileType_pkg.g_internal_error_msg);
122  end;
```

Остальные методы вызывают метод **WRITE_IO** с соответствующими параметрами:

```
124  member procedure close
125  is
126  begin
127     fileType.write_io(g_file_hdl, 'close');
128      filetype_pkg.g_files.delete(g_file_hdl);
129  end;
130
131  member procedure put(p_text in varchar2)
132  is
133  begin
134     fileType.write_io(g_file_hdL, 'put',p_text);
135  end;
136
137  member procedure new_line(p_lines in number default 1)
138  is
139  begin
140     fileType.write_io(g_file_hdl, 'new_line',p_lines);
141  end;
142
143  member procedure put_line(p_text in varchar2)
144  is
145  begin
146   fileType.write_io(g_file_hd1,   'put_line', p_text);
147  end;
148
149  member procedure putf
150  (p_fmt  in varchar2, p_arg1 in varchar2 default null,
151   p_arg2 in varchar2 default null, p_arg3 in varchar2 default null,
152   p_arg4 in varchar2 default null, p_arg5 in varchar2 default null)
153  is
154  begin
155      fileType.write_io
156    (g_file_hd1,   'putf', p_fmt, p_arg1,
157       p_arg2, p_arg3, p_arg4, p_arg5);
158  end;
159
160  member procedure flush
161  is
162  begin
163     fileType.write_io(g_file_hdl, 'flush');
164  end;
165
166  end;
```

```
167 /

Type body created.
```

В представленном коде перехватываются все исключительные ситуации пакета **UTL_FILE** и возбуждаются другие исключительные ситуации с помощью процедуры **RAISE_APPLICATION_ERROR**. Это основная причина инкапсуляции средств пакета **UTL_FILE** в объектный тип. Пакет **UTL_FILE** возбуждает исключительные ситуации с описанием **USER-DEFINED EXCEPTION**. Эти исключительные ситуации определены разработчиками пакета **UTL_FILE**, и при их возбуждении сервер Oracle выдает сообщение: "**USER-DEFINED EXCEPTION**". Оно не слишком информативно и не помогает разобраться в причинах ошибки. Я предпочитаю использовать процедуру **RAISE_APPLICATION_ERROR**, которая позволяет задать значения встроенных функций **SQLCODE** и **SQLERRM**, возвращаемые клиенту. Чтобы увидеть, как это может повлиять на отладку, достаточно рассмотреть следующий небольшой пример, демонстрирующий, какого рода сообщения об ошибках можно получить от пакета **UTL_FILE** и объектного типа **FILETYPE**:

```
tkyte@TKYTE816> declare
  2    futl_file.file_type :=utl_file.fopen('c:\temp\bogus',
  3                                         'foo.txt',  'w');
  4  begin
  5      utl_file.fclose(f);
  6  end;
  7  /
declare
*
ERROR at line 1:
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at "SYS.UTL_FILE", line 98
ORA-06512: at "SYS.UTL_FILE", line 157
ORA-06512: at line 2

tkyte@TKYTE816> declare
  2    f fileType := fileType.open('c:\temp\bogus',
  3                                'foo.txt',  'w');
  4  begin
  5      f.close;
  6  end;
  7  /
declare
*
ERROR at line 1:
ORA-20001: The path c:\temp\bogus is not in the utl_file_dir path
"c:\temp, c:\oracle"
ORA-06512: at "TKYTE.FILETYPE", line 54
ORA-06512: at line 2
```

Нетрудно понять, с помощью какого сообщения проще определить причину ошибки. Второе сообщение об ошибке (при наличии у владельца типа доступа к представлению **V$PARAMETER**) очень точно объясняет причину ошибки: использован недопус-

тимый каталог, не указанный в параметре инициализации **UTL_FILE_DIR**. Даже при отсутствии доступа к представлению **V$PARAMETER** будет выдано следующее сообщение:

```
*
ERROR at line 1:
ORA-20001: INVALID_PATH: File location or filename was invalid.
ORA-06512: at "TKYTE.FILETYPE", line 59
ORA-06512: at line 2
```

которое все равно лучше, чем краткое **user-defined exception.**

В этом типе также следует обратить внимание на возможность установки предпочтительных стандартных значений параметров подпрограмм. Например, до версии Oracle 8.0.5 пакет **UTL_FILE** имел ограничение на максимальную длину строки — 1023 байт. Если попытаться выдать более длинную строку, пакет **UTL_FILE** возбуждает исключительную ситуацию. По умолчанию точно так же происходит и в Oracle 8i. Если явно не указать максимальный размер строки при вызове **UTL_FILE.FOPEN,** ограничение 1023 байт остается. Я лично предпочитаю по умолчанию устанавливать максимальную длину строки равной 32 Кбайт. В начале кода я также задал стандартный режим открытия файла — на чтение ('R'). Поскольку в 90 процентах случаев я использую пакет UTL_FILE для чтения файла, такое стандартное значение для меня имеет смысл.

Теперь давайте проверим работу объектного типа и рассмотрим использование всех функций и процедур. Сначала создадим файл (предполагается, что мы работаем в Windows NT, существует каталог **c:\temp** и параметр инициализации **UTL_FILE_DIR** содержит **c:\temp)** и запишем в него определенные данные. Затем мы закроем этот файл, сохранив данные. Это продемонстрирует возможности записи типа **FILETYPE:**

```
tkyte@TKYTE816> declare
  2      f fileType := fileType.open('c:\terrp\ 'foo.txt', 'w');
  3  begin
  4      if (f.isOpen)
  5      then
  6      dbms_output.put_line('Файл      открыт');
  7      end if;
  8
  9      for i in 1 .. 10 loop
 10          f.put(i || ',');
 11      end loop;
 12      f.put_line(11);
 13
 14      f.new_line(5);
 15      for i in 1 .. 5
 16      loop
 17    f.put_line('строка    '    || i);
 18      end loop;
 19
 20      f.putf('%s %s', 'Hello', 'World');
 21
 22      f.flush;
 23
```

```
 24      f.close;
 25 end;
 26 /
Файл открыт

PL/SQL procedure successfully completed.
```

Далее продемонстрировано чтение файла с помощью объекта типа FILETYPE. Откроем только что записанный файл и убедимся, что прочитаны именно те данные, которые в нем содержатся:

```
tkyte@TKYTE816> declare
  2       ffileType :=fileType.open('c:\temp', 'foo.txt');
  3  begin
  4       if (f.isOpen)
  5      then
  6           dbms_output.put_line('Файл открыт');
  7       end if;
  8
  9      dbms_output.put_line
 10      ('строка 1: (должна быть 1,2,...,11)' || f.get_line);
 11
 12    for i in 2 .. 6
 13      loop
 14          dbms_output.put_line
 15        ('строка ' || i || ': (должна быть пустой)' || f.get_line);
 16      end loop;
 17
 18    for i in 7 .. 11
 19      loop
 20          dbms_output.put_line
 21        ('строка ' || to_char(i+1) ||
 22          ': (должна быть строка N)' || f.get_line);
 23      end loop;
 24
 25      dbms_output.put_line
 26     ('строка 12: (должна быть Hello World)' || f.get_line);
 27
 28      begin
 29          dbms_output.put_line(f.get_line);
 30          dbms_output.put_line('В предыдущей операторе должна
-> произойти ошибка');
 31      exception
 32          when NO_DATA_FOUND then
 33        dbms_output.put_line('получили       no data found, как и
-> ожидалось');
 34     end;
 35      f.close;
 36 end;
 37 /
Файл открыт
строка 1: (должна быть 1,2,...,11)1,2,3,4,5,6,7,8,9,10,11
```

```
строка 2: (должна быть пустой)
строка 3: (должна быть пустой)
строка 4: (должна быть пустой)
строка 5: (должна быть пустой)
строка 6: (должна быть пустой)
строка 8: (должна быть строка N)строка 1
строка 9: (должна быть строка N)строка 2
строка 10: (должна быть строка N)строка 3
строка 11: (должна быть строка N)строка 4
строка 12: (должна быть строка N)строка 5
строка 12: (должна быть Hello World)Hello World
получили no data found, как и ожидалось

PL/SQL procedure successfully completed.
```

Мы инкапсулировали пакет **UTL_FILE** в объектный тип Oracle. Получился замечательный интерфейс к стандартному пакету, работающий именно так, как нам хотелось. В терминах объектного программирования, мы создали класс **UTL_FILE,** включив в него методы, которые работают так, как нужно нам, а не так, как придумали разработчики Oracle. Мы не переписывали пакет **UTL_FILE,** а только создали для него другой интерфейс. Это хороший прием программирования, позволяющий легко решать проблемы, связанные с изменением реализации пакета **UTL_FILE** или появлением ошибки в новой версии. Все эти проблемы можно решить, изменив тело типа, а не сотни или тысячи непосредственно зависящих от пакета приложений. Например, в одной из версий пакета **UTL_FILE** нельзя было открыть несуществующий файл в режиме A (на добавление); при этом файл не создавался, хотя и должен был. Для решения проблемы достаточно было написать следующий код:

```
begin
    file_stat := utl_file.fopen(file_dir,filename,'a');
exception
    — если файл не существует, fopen не сработает
    — из-за ошибки режима 'a': 371510
    when utl_file.invalid_operation then
        — все остальные исключительные ситуации распространяются
        — во внешний блок, как обычно
        file_stat := utl_file.fopen(file_dir,file_name,'w');
end;
```

Если файлы открываются на добавление в 100 подпрограммах, придется делать много исправлений. Если же использован дополнительный уровень интерфейса, достаточно внести изменение только в одном месте.

Уникальные приемы использования наборов

Еще один вариант применения объектных типов в PL/SQL связан с использованием наборов и их возможностей взаимодействия с языками SQL и PL/SQL. Наборы позволяют сделать три вещи в SQL и PL/SQL, о реализации которых часто спрашивают разработчики.

- Как выбрать данные (SELECT *) из PL/SQL-функции?* Можно написать PL/SQL-функцию и обращаться с запросами к ней, а не к таблице базы данных.
- Как выбрать данные в массив записей? PL/SQL изначально позволяет выполнять выборку данных **BULK COLLECT** (выбирать по нескольку строк за раз) в PL/SQL-таблицу. К сожалению, это можно делать только в PL/SQL-таблицы с элементами **скалярных** типов. Я не могу сделать следующее:

  ```
  select c1, c2 BULK COLLECT INTO record_type from T
  ```

  поэтому приходится писать:

  ```
  select c1, c2 BULK COLLECT INTO table1, table2 from T
  ```

  С помощью наборов можно обеспечить выборку данных в записи.
- Как вставить запись? Вместо вставки по столбцам, я могу вставить в таблицу одну запись.

SELECT * из PL/SQL-функции

Чтобы продемонстрировать эту возможность, давайте вернемся к проблеме использования связываемых переменных (это моя любимая тема). Часто мне приходится слышать утверждения, что необходимо выполнить запрос вида:

```
select * from t where c in (:bind_variable)
```

где **BIND_VARIABLE** представляет собой список значений. Другими словами, переменная **BIND_VARIABLE** получает, допустим, значение '1, 2, 3' и необходимо, чтобы представленный выше запрос выполнялся как:

```
select * from t where c in (1, 2, 3)
```

и возвращал строки, в которых c = 1, 2 или 3; но при таком значении связываемой переменной фактически выполняется запрос:

```
select * from t where c in ('1,2,3')
```

Он возвращает строки, в которых c = '1,2,3' — значение столбца c равно одной такой строке. Такие запросы часто порождаются пользовательскими интерфейсами, в которых пользователь может отметить флажками одно или несколько (любое количество) возможных значений из списка. Чтобы не создавать уникальные запросы для каждого сочетания флажков (мы знаем, насколько это плохо), необходим метод связывания произвольного количества элементов в списке. Ну, поскольку можно выполнять SELECT * из PL/SQL-функции, решение есть. Сейчас я его продемонстрирую:

```
tkyte@TKYTE816> create or replace type myTableType
  2  as table of number;
  3  /

Type created.
```

Созданный таким образом тип данных и будет возвращать PL/SQL-функция. Этот тип **обязательно** должен быть задан на уровне SQL с помощью оператора **CREATE TYPE.**

** Другая формулировка той же проблемы: как возвращать результирующие множества из хранимых функций? Прим. научн. ред.*

Нельзя использовать тип, заданный в PL/SQL-пакете; конечная цель — выбирать данные с помощью SQL-операторов, поэтому необходим тип данных SQL. Этот пример также показывает, что я не являюсь принципиальным противником использования вложенных таблиц. Именно этот тип набора оптимален при программировании на PL/SQL, если приходится решать подобного рода задачи. При использовании массива VARRAY придется искусственно ограничить количество элементов. Размер же вложенной таблицы ограничен только объемом памяти, доступной в системе.

```
tkyte@TKYTE816>create or replace
  2 function str2tbl(p_str in varchar2) return myTableType
  3 as
  4     l_str   long default p_str ||  ',';
  5      l_n         number;
  6     l_data    myTableType := myTabletype();
  7 begin
  8      loop
  9      l_n := instr(l_str, ' , ' ) ;
 10        exit when (nvl(l_n,0) = 0);
 11         l_data.extend;
 12        l_data(l_data.count)  :=
 13             ltrim(rtrim(substr(l_str,1,l_n-1)));
 14        l_str := substr(l_str, l_n+1);
 15      end loop;
 16     return l_data;
 17 end;
 18  /

Function created.
```

Итак, создана PL/SQL-функция, принимающая строку со списком значений через запятую и преобразующая ее в SQL-тип **MYTABLETYPE**. Осталось только найти способ выбрать эти данные с помощью SQL-оператора. Это легко сделать с помощью оператора TABLE и преобразования типа, CAST:

```
tkyte@TKYTE816>variablebind_variablevarchar2(30)
tkyte@TKYTE816>exec :bind_variable := '1,3,5,7,99'

PL/SQL procedure successfully completed.

BIND_VARIABLE

1,3,5,7,99

tkyte@TKYTE816> select *
  2   from TABLE (cast(str2tbl(:bind_variable) asmyTableType))
  3  /

COLUMN_VALUE

           1
           3
           5
           7
          99
```

Теперь легко использовать эту конструкцию в подзапросе IN:

```
tkyte@TKYTE816> select *
  2    from all_users
  3   where user_id in
  4       (select *
  5            from TABLE (cast(str2tbl(:bind_variable) as myTableType))
  6       )
  7  /
```

```
USERNAME                       USER_ID CREATED

SYSTEM                               5 04-NOV-00
```

Этот прием можно использовать во многих случаях. Можно применить к значению PL/SQL-переменной типа набора операцию **ORDER BY**, можно возвращать клиенту наборы данных, сгенерированные PL/SQL-функцией, можно задавать конструкции WHERE для выбора определенных значений PL/SQL-переменных и т.д.

Если пойти чуть дальше, таким способом можно возвращать полные результирующие множества с несколькими столбцами. Например:

```
tkyte@TKYTE816>  create type myRecordType as object
  2  (seq int,
  3   a int,
  4   b varchar2(10),
  5   c date
  6  )
  7  /

Type created.

tkyte@TKYTE816> create table t (x int, y varchar2(10), z date);
Table created.

tkyte@TKYTE816> create or replace type myTableType
  2  as table of myRecordType
  3  /

Type created.

tkyte@TKYTE816> create or replace function my_function return myTableType
  2  is
  3      l_data myTableType;
  4  begin
  5     l_data := myTableType();
  6
  7     for i in 1..5
  8     loop
  9         l_data.extend;
 10       l_data(i) :=myRecordType(i, i, 'row ' || i, sysdate+i);
 11     end loop;
 12     return l_data;
 13  end;
 14  /

Function created.
```

```
tkyte@TKYTE816> select *
  2    from TABLE (cast(my_function() as mytableType))
  3   where c > sysdate+1
  4   order by seq desc
  5  /
```

SEQ	A	B	C
5	5	row 5	29-MAR-01
4	4	row 4	28-MAR-01
3	3	row 3	27-MAR-01
2	2	row 2	26-MAR-01

Множественная выборка данных в записи

Итак, мы рассмотрели, как использовать наборы для выборки данных из PL/SQL-функции. Теперь разберемся, как с их помощью обеспечить множественную выборку данных в аналог PL/SQL-записей. Выполнить множественную выборку в массив реальных PL/SQL-записей нельзя, но можно выбрать данные во вложенную таблицу SQL. Для этого потребуется два объектных типа: скалярный тип, представляющий запись, и вложенная таблица с записями этого типа. Например:

```
tkyte@TKYTE816> create type myScalarType
  2  as object
  3  (username varchar2(30),
  4   user_id  number,
  5   created  date
  6  )
  7  /

Type created.

tkyte@TKYTE816> create type myTableType as table of myScalarType
  2  /

Type created.
```

Теперь все готово для выборки данных в переменную типа MYTABLETYPE следующим образом:

```
tkyte@TKYTE816> declare
  2      l_users    myTableType;
  3  begin
  4      select cast(multiset(select username, user_id, created
  5                             from all_users
  6                            order by username)
  7                  as myTableType)
  8        into l_users
  9        from dual;
 10
 11      dbms_output.put_line('Retrieved '|| l_users.count || ' rows');
 12  end;
 13  /
Retrieved 25 rows

PL/SQL procedure successfully completed.
```

Запрос к представлению ALL_USERS можно заменить любым запросом, выбирающим строку типа VARCHAR2(30), число и дату. Запрос может быть сколь угодно сложным, включать соединения и т.п. Фокус в том, что результаты этого подзапроса преобразуются в объектный тип. Затем можно выбрать все результирующее множество в локальную переменную с помощью стандартного оператора **SELECT ... INTO**.

Вставка записей

Зная, что можно выполнять операторы **SELECT * FROM НАБОР**, где **НАБОР** — это либо локальная переменная, либо PL/SQL-функция, возвращающая вложенную таблицу, нетрудно придумать, как выполнить аналогичным образом оператор INSERT. Необходимо задать переменную типа вложенной таблицы и заполнить ее записями, которые требуется вставить. Следующий пример демонстрирует, как будет выглядеть вставка одной строки:

```
tkyte@TKYTE816> create table t as select * from all_users where 1=0;

Table created.

tkyte@TKYTE816> declare
  2      l_users    myTableType :=
  3                     myTableType(myScalarType('tom', 1, sysdate));
  4  begin
  5          insert into t
  6          select * from TABLE (cast(l_users as myTableType));
  7  end;
  8  /
tkyte@TKYTE816> select * from t;

USERNAME            USER_ID CREATED

tom                       1 24-MAR-01
```

При работе с много-столбцовой таблицей этот прием может пригодиться.

Итак, в этом разделе мы рассмотрели использование объектных типов Oracle для расширения возможностей языка PL/SQL аналогично тому, как для этих целей используются классы в языках Java или C++.

Описаны также интересные варианты использования вложенных таблиц. Возможность выполнять оператор SELECT * из PL/SQL-функции открывает заманчивые перспективы. Списки значений произвольной длины для конструкции IN — только начало. Возможности поистине безграничны. Можно написать небольшую функцию, использующую средства пакета UTL_FILE для чтения файла ОС, разбиения прочитанных строк на поля по запятым и возвращения результирующего множества, построенного по содержимому обычного файла, для вставки в другую таблицу или соединения с существующей таблицей.

Такое использование объектных типов дает новую жизнь хорошо зарекомендовавшему себя языку программирования. Создав самостоятельно один-два типа, вы найдете применение этой методике во множестве приложений. Это логический способ объединения данных и функций для работы с ними, что является одной из основных целей объектно-ориентированного программирования. Предвидя протесты, я не называю это

чисто объектно-ориентированным программированием на PL/SQL, но это, определенно, очень близкая к нему методика.

Объектно-реляционные представления

Это весьма мощное средство для тех, кто хочет работать с объектно-реляционными расширениями, но должен обеспечивать для множества приложений реляционное представление данных. Можно использовать стандартный механизм **VIEW** для создания объектов на основе реляционных таблиц. Не нужно создавать объектные таблицы, со всеми их мистическими столбцами — можно создать объектное представление стандартных таблиц (скорее всего уже существующих). Подобные представления обеспечат возможности, аналогичные объектной таблице того же типа, но без дополнительных затрат ресурсов на поддержку скрытых ключей, суррогатных ключей и пр.

В этом разделе мы используем таблицы **EMP** и **DEPT** для создания представления данных по отделам. Это очень похоже на пример создания вложенной таблицы в главе 6, где был создан тип **EMP_TAB_TYPE** как вложенная таблица записей типа **EMP_TYPE,** а таблица **DEPT** содержала столбец — вложенную таблицу этого типа. Здесь мы еще раз смоделируем типы **EMP_TYPE** и **EMP_TAB_TYPE,** но создадим еще и объектный тип **DEPT_TYPE,** а также представление этого типа.

Интересно отметить, что использование объектных представлений позволяет взять лучшее из двух миров (реляционного и объектно-реляционного). Например, можно создать приложение, в котором должно использоваться представление данных по отделам. В представлении указаны данные по отделам, а информация о сотрудниках отдела естественным образом представляется как набор, один из атрибутов отдела. Другому приложению необходимо другое представление тех же данных. Например, службе охраны на проходной необходим доступ к данным по сотрудникам. Отдел в данном случае - лишь атрибут записи о сотруднике, в то время как для другого приложения список сотрудников является атрибутом отдела. В этом — сила реляционной модели: она позволяет одновременно поддерживать несколько представлений данных. Объектная модель не обеспечивает поддержку нескольких различных представлений одних и тех же данных так же просто (если вообще обеспечивает) или эффективно. Используя несколько различных объектных представлений реляционных данных, можно обеспечить потребности всех приложений.

Необходимые типы

Использованные в этом примере типы позаимствованы из главы 6, с добавлением типа **DEPT_TYPE.** Вот как они создаются:

```
scott@TKYTE816> create or replace type emp_type
  2  as object
  3   (empno       number(4),
  4    ename       varchar2(10),
  5    job         varchar2(9),
  6    mgr         number(4),
  7    hiredate    date,
  8    sal        number(7, 2),
```

```
  9   comm        number(7, 2)
 10 );
 11 /

Type created.

scott@TKYTE816>create or replace type emp_tab_type
  2 as table of emp_type
  3 /

Type created.

scott@TKYTE816>create or replace type dept_type
  2 as object
  3  (deptno number(2),
  4   dname  varchar2(14),
  5   loc       varchar2(13),
  6   emps      emp_tab_type
  7 )
  8 /

Type created.
```

Отдел моделируется как объект с такими атрибутами: номер отдела (deptno), название (dname), местонахождение (loc) и список сотрудников (emps).

Объектно-реляционное представление

По представленным выше определениям типов легко получить данные для этого представления по существующим реляционным данным. Вот как создается представление:

```
scott@TKYTE816>create or replace view dept_or
  2 of dept_type
  3 with object identifier(deptno)
  4 as
  5 select deptno, dname, loc,
  6        cast (multiset (
  7              select empno, ename, job, mgr, hiredate, sal, comm
  8                   from emp
  9                  where emp.deptno = dept.deptno)
 10               as emp_tab_type )
 11    from dept
 12 /

View created.
```

Назначение конструкций **CAST** и **MULTISET** вам уже знакомо — с их помощью коррелированный подзапрос преобразуется в набор типа вложенной таблицы. Для каждой строки в таблице **DEPT** мы запрашиваем всех сотрудников соответствующего отдела. С помощью конструкции **WITH OBJECT IDENTIFIER** серверу Oracle можно сообщить, какой столбец (или столбцы) однозначно идентифицируют строку в представлении. Это позволяет серверу автоматически создать ссылку на объект (object reference), что и обеспечивает возможности работы с этим представлением как с объектной таблицей.

Созданное представление можно использовать:

```
scott@TKYTE816> select dname, d.emps
  2    from dept_or d
  3  /
```

```
DNAME          EMPS(EMPNO, ENAME, JOB, MGR, HIREDATE, S

ACCOUNTING     EMP_TAB_TYPE(EMP_TYPE(7782, 'CLARK',
                'MANAGER', 7839, '09-JUN-81', 2450,
                NOLL), EMP_TYPE(7839, 'KING',
                'PRESIDENT', NULL, '17-NOV-81', 5000,
                NULL), EMP_TYPE(7934, 'MILLER', 'CLERK',
                7782, '23-JAN-82', 1300, NULL))

RESEARCH       EMP_TAB_TYPE(EMP_TYPE(7369, 'SMITH',
                'CLERK', 7902, '17-DEC-80', 800, NULL),
                EMP_TYPE(7566, 'JONES', 'MANAGER', 7839,
                '02-APR-81', 2975, NULL), EMP_TYPE(7788,
                'SCOTT', 'ANALYST', 7566, '09-DEC-82',
                3000, NULL), EMP_TYPE(7876, 'ADAMS',
                'CLERK', 7788, '12-JAN-83', 1100, NULL),
                EMP_TYPE(7902, 'FORD', 'ANALYST', 7566,
                '03-DEC-81', 3000, NULL))

SALES          EMP_TAB_TYPE(EMP_TYPE(7499, 'ALLEN',
                'SALESMAN', 7698, '20-FEB-81', 1600,
                300), EMP_TYPE(7521, 'WARD', 'SALESMAN',
                7698, '22-FEB-81', 1250, 500),
                EMP_TYPE(7654, 'MARTIN', 'SALESMAN',
                7698, '28-SEP-81', 1250, 1400),
                EMP_TYPE(7698, 'BLAKE', 'MANAGER', 7839,
                '01-MAY-81', 2850, NULL), EMP_TYPE(7844,
                'TURNER', 'SALESMAN', 7698, '08-SEP-81',
                1500, 0), EMP_TYPE(7900, 'JAMES',
                'CLERK', 7698, '03-DEC-81', 950, NULL))

OPERATIONS     EMP_TAB_TYPE()

4 rows selected.
```

```
scott@TKYTE816> select deptno, dname, loc, count(*)
  2    from dept_or d, table (d.emps)
  3   group by deptno, dname, loc
  4  /
```

```
    DEPTNO DNAME          LOC            COUNT(*)

        10 ACCOUNTING     NEW YORK              3
        20 RESEARCH       DALLAS                5
        30 SALES          CHICAGO               6

3 rows selected.
```

Итак, у нас есть реляционные таблицы и объектно-реляционное представление. Пользователю трудно определить, где — представление, а где — таблицы. Все возмож-

ности объектной таблицы доступны: из нее можно выбрать ссылки на объекты, вложенная таблица создана и т.д. Преимущество такой реализации в том, что мы явно указываем, как соединять таблицы EMP и DEPT, используя естественное отношение — главный/подчиненный.

Итак, создано объектно-реляционное представление, позволяющее запрашивать данные. Но изменять его данные пока нельзя:

```
scott@TKYTE816> update TABLE (select p.emps
  2                   from dept_or p
  3                  where deptno = 20)
  4     set ename = lower(ename)
  5  /
   set ename = lower(ename)
       *
ERROR at line 4:
ORA-25015: cannot perform DML on this nested table view column

scott@TKYTE816> declare
  2     l_emps  emp_tab_type;
  3  begin
  4      select p.emps into l_emps
  5        from dept_or  p
  6       where deptno = 10;
  7
  8      for i in 1 .. l_emps.count
  9       loop
 10         l_emps(i).ename := lower(l_emps(i).ename);
 11       end loop;
 12
 13      update dept_or
 14         set emps = l_emps
 15       where deptno = 10;
 16  end;
 17  /
declare
*
ERROR at line 1:
ORA-01733: virtual column not allowed here
ORA-06512: at line 13
```

Необходимо заставить представление "обновляться". У нас реализовано сложное сопоставление реляционных данных объектно-реляционным (на самом деле оно может иметь любую степень сложности). Так как же заставить представление "обновляться"? Сервер Oracle обеспечивает для этого соответствующий механизм — триггеры **INSTEAD** 0F. Можно реализовать алгоритм, который должен выполняться сервером Oracle вместо (**INSTEAD OF**) стандартного при изменении содержимого представления. Чтобы продемонстрировать это, давайте обеспечим возможность изменения показанного ранее представления.

Сервер Oracle позволяет создать триггер **INSTEAD OF** по представлению **DEPT_OR** и по любому типу вложенной таблицы, входящей в это представление. Создав триггер

по столбцам вложенной таблицы, можно изменять столбец вложенной таблицы так, будто это обычная таблица. Соответствующий триггер может иметь следующий вид:

```
scott@TKYTE816> create or replace trigger EMPS_IO_UPDATE
  2  instead of UPDATE on nested table emps of dept_or
  3  begin
  4       if (:new.empno = :old.empno)
  5       then
  6           update emp
  7             set ename = :new.ename, job = :new.job, mgr = :new.mgr,
  8              hiredate = :new.hiredate, sal = :new.sal,
-> comm = :new.comm
  9           where empno = :old.empno;
 10       else
 11       raise_application_error(-20001,'Значение      столбца empno
-> изменять нельзя');
 12       end if;
 13  end;
 14  /

Trigger created.
```

Как видите, триггер будет срабатывать при изменении (**INSTEAD OF UPDATE**) столбца типа вложенной таблицы, **EMPS**, представления **DEPT_OR**. Он будет срабатывать для каждой изменяемой строки вложенной таблицы и имеет доступ к значениям до и после изменения (**:OLD** и **:NEW**), как и "обычный" триггер. В данном случае понятно, что надо делать, — изменить строку таблицы **EMP** с соответствующим значением **EMPNO**, задав ее столбцам новые значения. В этом триггере я принудительно запрещаю изменять первичный ключ (мы используем объектно-реляционные средства, но это не значит, что можно нарушать основные принципы проектирования реляционных баз данных).

Теперь, выполним следующие операторы:

```
scott@TKYTE816> update TABLE (select p.emps
  2                    from dept_or p
  3                   where deptno = 20)
  4     set ename = lower(ename)
  5  /

5 rows updated.

scott@TKYTE816> select ename from emp where deptno = 20;

ENAME

smith
jones
scott
adams
ford

scott@TKYTE816> select ename
  2    from TABLE(select p.emps
  3                    from dept_or p
  4                   where deptno = 20);
```

```
ENAME

smith
jones
scott
adams
ford
```

Как видите, изменение вложенной таблицы успешно преобразовано в изменение реляционной таблицы, как и ожидалось. Так же легко написать триггеры на события **INSERT** и **DELETE**, поскольку **UPDATE** — самый сложный случай. Так что на этом и остановимся.

Если сейчас выполнить следующее изменение:

```
scott@TKYTE816> declare
  2    l_emps  emp_tab_type;
  3  begin
  4     select p.emps into l_emps
  5       from dept_or  p
  6      where deptno = 10;
  7
  8     for i in 1 .. l_emps.count
  9      loop
 10        l_emps(i).ename := lower(l_emps(i).ename);
 11      end loop;
 12
 13     update dept_or
 14        set emps = l_emps
 15      where deptno = 10;
 16  end;
 17  /
declare
*
ERROR at line 1:
ORA-01732: data manipulation operation not legal on this view
ORA-06512: at line 13
```

то окажется, что при выполнении выдается сообщение об ошибке. Странно. Не должен ли сработать созданный ранее триггер? На самом деле — нет. Только изменения вложенной таблицы, связанные с извлечением ее данных, вызовут срабатывание триггера. Тригтер срабатывает, только если с вложенной таблицей работают как с обычной таблицей. Мы же не выполняем операцию над большим количеством данных вложенной таблицы, а только изменяем столбец представления **DEPT_OR.** Чтобы поддержать работу подобного кода (и изменений других скалярных столбцов представления **DEPT_OR),** необходимо создать триггер **INSTEAD OF** для представления **DEPT_OR.** Этот триггер будет обрабатывать значения **:OLD.EMPS** и **:NEW.EMPS** как множества следующим образом.

1. Удалять из таблицы **EMP все** записи, значение **EMPNO** которых было в наборе **:OLD,** но отсутствует в наборе **:NEW.** Для этого прекрасно подходит реляционный оператор **MINUS.**

2. зменять в таблице **EMP** каждую запись, значение столбца **EMPNO** которой входит во множество значений EMPNO, соответствующие :**NEW**-записи которых отличаются от :**OLD**-записей. Это множество легко найти с помощью оператора **MINUS**.
3. Вставлять в таблицу **EMP** все записи :**NEW**, значение :**NEW.EMPNO** которых не находится во множестве значений **EMPNO** :**OLD**-записи.

Вот как это реализуется:

```
scott@TKYTE816> create or replace trigger DEPT_OR_IO_UPDATE
  2  instead of update on dept_or
  3  begin
  4      if (:new.deptno = :old.deptno)
  5      then
  6          if updating('DNAME') or updating('LOC')
  7          then
  8             update dept
  9               set dname = :new.dname, loc = :new.loc
 10              where deptno = :new.deptno;
 11          end if;
 12
 13          if (updating('EMPS'))
 14          then
 15             delete from emp
 16              where empno in
 17               (select empno
 18                   from TABLE(cast(:old.emps as emp_tab_type))
 19                  MINUS
 20                   select empno
 21                     from TABLE(cast(:new.emps as emp_tab_type))
 22               );
 23      dbms_output.put_line('удалено       ' || sql%rowcount);
```

Первый оператор MINUS возвращает множество значений EMPNO, которые были в наборе :OLD, но отсутствуют в наборе :NEW. Эти записи надо удалить из таблицы EMP, поскольку в наборе их больше нет. Изменим те записи набора, которые были обновлены:

```
 24
 25             update emp E
 26               set (deptno, ename, job, mgr,
 27                    hiredate, sal, comm) =
 28                   (select :new.deptno, ename, job, mgr,
 29                           hiredate, sal, comm
 30                      from TABLE(cast(:new.emps as emp_tab_type)) T
 31                     where T.empno = E.empno
 32                    )
 33              where empno in
 34                 (select empno
 35                     from (select *
 36                             from TABLE(cast(:new.emps as emp_tab_type))
 37                             MINUS
 38                            select *
```

```
39                                  from TABLE(cast(:old.emps as emp_tab_type))
40                              )
41                  );
42        dbms_output.put_line('изменено       ' || sql%rowcount);
```

Этот оператор **MINUS** возвращает все значения из :**NEW**, кроме совпадающих со значениями из :**OLD**; это дает множество измененных записей. Оно используется в подзапросе для получения множества значений **EMPNO**, соответствующие записи для которых в таблице **EMP** надо изменить, после чего эти значения определяются с помощью коррелированного подзапроса. Наконец, добавляем новые записи:

```
43
44              insert into emp
45             (deptno, empno, ename, job, mgr, hiredate, sal, comm)
46              select  :new.deptno,empno,ename,job,mgr,hiredate,sal,comm
47                from (select *
48                         from TABLE(cast(:new.emps as emp_tab_type))
49                        where empno in
50                           (select empno
51                               from TABLE(cast(:new.emps as
-> emp_tab_type))
52                              MINOS
53                             select empno
54                               from TABLE(cast(:old.emps as
-> emp_tab_type))
55                           )
56                    );
57       dbms_output.put_line('вставлено       ' II sql%rowcount);
58          else
59        dbms_output.put_line('обработка       вложенной таблицы
-> пропущена');
60          end if;
61      else
62           raise_application_error(-20001,'значение deptno изменять
-> нельзя');
63      end if;
64  end;
65  /

Trigger created.
```

Оператор **MINUS** генерирует множество значений **EMPNO** из набора **:NEW,** отсутствующих в наборе **:OLD;** оно представляет список строк, которые надо добавить в таблицу **EMP.**

Триггер кажется огромным, но на самом деле он простой. Вначале он проверяет, не изменены ли скалярные столбцы представления **DEPT_OR.** Если — да, выполняются соответствующие изменения в таблице **DEPT.** Затем, если был изменен столбец вложенной таблицы (заменены все ее значения), эти изменения вносятся в таблицу **EMP.** Чтобы внести все необходимые изменения, надо:

1. удалить из таблицы **EMP** записи, которые были удалены из столбца вложенной таблицы EMPS;

2. изменить в таблице **EMP** те записи, значения которых были изменены в столбце вложенной таблицы **EMPS**;
3. вставить в таблицу **EMP** записи, добавленные в столбец вложенной таблицы EMPS.

К счастью, SQL-оператор **MINUS** и возможность преобразовать столбец типа вложенной таблицы в реальную таблицу упрощает реализацию триггера. Теперь мы можем обрабатывать данные так:

```
scott@TKYTE816> declare
  2     l_emps  emp_tab_type;
  3  begin
  4      select p.emps into l_emps
  5        from dept_or  p
  6       where deptno = 10;
  7
  8      for i in 1 .. l_emps.count
  9      loop
 10         l_emps(i).ename := lower(l_emps(i).ename);
 11      end loop;
 12
 13      update dept_or
 14         set emps = l_emps
 15       where deptno = 10;
 16  end;
 17  /
удалено 0
изменено 3
вставлено 0

PL/SQL procedure successfully completed.

scott@TKYTE816> declare
  2     l_emps  emp_tab_type;
  3  begin
  4      select p.emps into l_emps
  5        from dept_or  p
  6       where deptno = 10;
  7
  8
  9      for i in 1 .. l_emps.count
 10      loop
 11          if (l_emps(i).ename = 'miller')
 12          then
 13              l_emps.delete(i);
 14          else
 15              l_emps(i).ename := initcap(l_emps(i).ename);
 16          end if;
 17      end loop;
 18
 19      l_emps.extend;
 20      l_emps(l_emps.count) :=
 21        emptype(1234, 'Tom', 'Boss',
```

```
 22                null, sysdate, 1000, 500);
 23
 24      update dept_or
 25         set emps = l_emps
 26       where deptno = 10;
 27  end;
 28  /
удалено 1
изменено 2
вставлено 1

PL/SQL procedure successfully completed.
scott@TKYTE816> update dept_or set dname = initcap(dname);
Обработка вложенной таблицы пропущена
Обработка вложенной таблицы пропущена
Обработка вложенной таблицы пропущена

Обработка вложенной таблицы пропущена

4 rows updated.

scott@TKYTE816> commit;

Commit complete.
```

Триггер преобразует наши действия с экземпляром объекта в соответствующие изменения базовых реляционных таблиц.

Возможность работать с реляционными данными через объектно-реляционные представления позволяет максимально использовать преимущества как реляционной, так и объектно-реляционной модели.

Реляционная модель сильна тем, что с ее помощью можно ответить практически на любой вопрос о базовых данных. Рассматриваются ли данные по отделам (надо запросить информацию об отделе и его сотрудниках) или по сотрудникам (надо задать номер сотрудника и получить информацию об отделе, в котором он работает), всегда есть возможность выполнить соответствующий запрос. Реляционные таблицы можно использовать непосредственно или сгенерировать модель на основе объектного типа — все необходимые данные будут объединены и представлены в удобном для использования виде. Рассмотрим результаты следующих запросов:

```
scott@TKYTE816> select * from dept_or where deptno = 10;

   DEPTNO DNAME        LOC            EMPS(EMPNO, ENAME, JOB, MGR,
HIREDATE, S

       10 Accounting   NEW YORK       EMP_TAB_TYPE(EMP_TYPE(7782, 'Clark',
                                       •MANAGER[1], 7839, '09-JUN-81', 2450,
                                      NULL), EMP_TYPE(7839, 'King',
                                       'PRESIDENT', NULL, '17-NOV-81', 5000,
                                      NULL), EMP_TYPE(1234, 'Tom', 'Boss',
                                      NULL, '25-MAR-01', 1000, 500))
scott@TKYTE816> select dept.*, empno, ename, job, mgr, hiredate, sal, comm
  2  from emp, dept
  3  where emp.deptno = dept.deptno
  4  and dept.deptno = 10
  5  /
```

```
DEPTNO DNAME      LOC      EMPNO ENAME JOB       MGR  HIREDATE    SAL  COM

    10 Accounting NEW YORK  7782 Clark MANAGER  7839 09-JUN-81 2450
    10 Accounting NEW YORK  7839 King  PRESIDENT     17-NOV-81 5000
    10 Accounting NEW YORK  1234 Tom   Boss          25-MAR-01 1000  500
```

Они возвращают похожие данные. Первый запрос в сжатом виде выдает всю информацию об отделе в виде одной строки. Он может возвращать несколько вложенных таблиц, для чего в классическом языке SQL пришлось бы выполнить несколько запросов.

На сервере можно выполнить много действий, формируя ответ и возвращая его в виде одной строки. Когда работа выполняется в среде, где лишних обменов данными по сети следует избегать (из-за большого времени ожидания), это дает существенные преимущества. Не говоря уже о том, что один простой оператор **SELECT * FROM T** может выполнить действия нескольких SQL-операторов. Учтите также, что в объектном представлении нет повторяющихся столбцов данных. Значения столбцов **DEPTNO**, **DNAME** и **LOC** не повторяются для каждого сотрудника; они возвращаются только один раз, что для многих приложений более удобно.

Второй запрос требует от разработчика более глубокого знания данных, и эту особенность следует учитывать. Надо знать, как соединять данные; если же соединяется много таблиц, возможно, потребуются дополнительные запросы, результаты которых придется обрабатывать самостоятельно. Поясню. Предположим, в модели необходимо учесть бюджет отдела на финансовый год. Он хранится в следующей реляционной таблице:

```
scott@TKYTE816>create table dept_fy_budget
  2  (deptno  number(2) references dept,
  3   fy       date,
  4   amount   number,
  5  constraint dept_fy_budget_pkprimary key(deptno,fy)
  6  )
  7  /

Table created.
```

В этой таблице хранятся данные о бюджете отделов за этот и несколько предыдущих финансовых лет. Для работы приложения необходимо представление данных по отделам, включающее все скалярные данные (название, местонахождение). Кроме того, необходима информация о сотрудниках (столбец типа **EMP_TAB_TYPE**), а также данные о бюджете за последние финансовые годы. Для получения этих данных по реляционной модели разработчику приложения придется выполнять следующие операторы:

```
scott@TKYTE816> select dept.*, ettpno, ename, job, mgr, hiredate, sal, comn
  2  from emp, dept
  3 where emp.deptno = dept.deptno
  4  and dept.deptno = 10
  5  /
DEPTNO DNAME      LOC      EMPNO ENAME JOB       MGR  HIREDATE    SAL  COM

    10 Accounting NEW YORK  7782 Clark MANAGER  7839 09-JON-81  2450
    10 Accounting NEW YORK  7839 King  PRESIDENT    17-NOV-81  5000
    10 Accounting NEW YORK  1234 Tom   Boss         25-MAR-01  1000  500
```

```
3 rows selected.
scott@TKYTE816> select fy, amount
  2  from dept_fy_budget
  3  where deptno = 10
  4  /

FY              AMOUNT

01-JAN-99          500
Ol-JAN-00          750
01-JAN-01         1000

3 rows selected.
```

Нельзя написать **один** реляционный запрос, выдающий все эти данные. Можно использовать ряд расширений Oracle (функцию **CURSOR в** SQL) для возврата строк, каждая из которых содержит результирующее множество:

```
scott@TKYTE816> select
  2 dept.deptno, dept.dname,
  3 cursor(select empno from emp where deptno = dept.deptno),
  4 cursor(select fy, amount from dept_fy_budget where deptno =
                                                      dept.deptno)
  5   from dept
  6  where deptno = 10
  7  /

DEPTNO DNAME          CURSOR(SELECTEMPNOFR CURSOR(SELECTFY,AMOU

    10 ACCOUNTING     CURSOR STATEMENT : 3 CURSOR STATEMENT : 4

CURSOR STATEMENT : 3

     EMPNO

      7782
      7839
      7934

3 rows selected.

CURSOR STATEMENT : 4

FY              AMOUNT

01-JAN-99          500
Ol-JAN-00          750
Ol-JAN-01         1000

3 rows selected.

1 row selected.
```

В данном случае была выбрана одна строка, вернувшая клиенту еще два курсора. Клиентское приложение выбрало данные из этих двух курсоров и выдало результаты. Это прекрасно работает, но требует знания особенностей базовых данных и способов

их объединения (как написать коррелированные подзапросы для генерации курсоров). Эту модель можно реализовать с помощью объектно-реляционных расширений, пересоздав представление следующим образом:

```
scott@TKYTE816> create or replace type dept_budget_type
  2  as object
  3   (fy        date,
  4    amount number
  5   )
  6   /

Type created.

scott8TKYTE816> create or replace type dept_budget_tab_type
  2  as table of dept_budget_type
  3  /

Type created.

scott@TKYTE816> create or replace type dept_type
  2  as object
  3  (deptno number(2),
  4   dname  varchar2(14),
  5    loc    varchar2(13),
  6    emps   emp_tab_type,
  7    budget dept_budget_tab_type
  8  )
  9  /

Type created.

scott6TKYTE816> create or replace view dept_or
  2  of dept_type
  3  with object identifier(deptno)
  4  as
  5  select deptno, dname, loc,
  6         cast  (multiset  (
  7               select empno, ename, job, mgr, hiredate, sal, conm
  8                   from emp
  9                  where emp.deptno = dept.deptno)
 10               as emp_tab_type)  emps,
 11         cast  (multiset  (
 12                 select fy, amount
 13                    from dept_fy_budget
 14                   where dept_fy_budget.deptno = dept.deptno)
 15               as dept_budget_tab_type)  budget
 16    from dept
 17  /
View created.
```

Теперь учтите, что представленные выше действия выполняются один раз, и все сложности от приложения скрыты. В приложении можно просто написать:

```
scott@TKYTE816> select * from dept_or where deptno = 10
  2  /
```

```
DEPTNO DNAME      LOC      EMPS(EMPNO, ENAME, J BUDGET(FY, AMOUNT)

    10 Accounting NEW YORK EMP_TAB_TYPE(EMP_TYP DEPT_BUDGET_TAB_TYPE
                           E(7782, 'Clark',      (DEPT_BUDGET_TYPE('0
                           'MANAGER', 7839,  1-JAN-99',  500),
                           '09-JUN-81', 2450,    DEPT_BUDGET_TYPE('01
                           NULL),                -JAN-00', 750),
                           EMP_TYPE(7839,      DEPT_BUDGET_TYPE('01
                           'King', 'PRESIDENT', -JAN-01', 1000))
                           NULL, '17-NOV-81',
                           5000, NULL),
                           EMP_TYPE(1234,
                           'Tom', 'Boss', NULL,
                           '25-MAR-01', 1000,
                           500))

1 row selected.
```

Снова мы получаем одну строку, один экземпляр объекта, представляющего данные в нужном виде. Это весьма удобно. Сложность базовой физической модели скрыта, и легко понять, как создать графический интерфейс для представления этих данных пользователю. При программировании на языке Java (с помощью интерфейса JDBC), Visual Basic (с помощью объектов OO4O — Oracle Objects for Ole), PL/SQL, использовании библиотеки OCI (Oracle Call interface) и прекомпилятора Pro*C объектно-реляционные расширения легко применить. Использовать реляционную модель становится все труднее по мере добавления сложных отношений один ко многим. При использовании объектно-реляционной модели все несколько проще. Придется, конечно, изменить триггеры INSTEAD OF для поддержки изменения базовых реляционных данных, так что приведенный пример неполон, но идею вы, надеюсь, уловили.

Резюме

В этой главе мы изучили основные способы использования объектных типов и расширений сервера Oracle. Из четырех возможных способов мы детально рассмотрели три.

Мы рассмотрели использование объектных типов для расширения стандартного набора типов системы. С помощью нового типа **ADDRESS_TYPE** мы смогли не только задать общую систему именования и использования адресов, но и обеспечить специализированные методы и средства работы с ними.

Мы также рассмотрели использование объектно-реляционных расширений для естественного развития возможностей языка PL/SQL. Мы взяли стандартный пакет и реализовали для него интерфейс на уровне объектного типа. Это позволяет защититься от изменений в реализации стандартного пакета, а также обеспечивает более "объектно-ориентированный" стиль программирования на PL/SQL, подобный использованию классов в языках C++ или Java. Кроме того, было показано, как, используя наборы, можно выбирать данные из PL/SQL-функции с помощью оператора SELECT. Одной этой возможности достаточно, чтобы оправдать использование объектно-реляционных расширений.

Наконец, мы изучили, как использовать эти средства для создания объектно-реляционных представлений реляционных, по сути, данных. Как оказалось, можно легко создать специализированные объектные представления реляционных данных для любого количеств различных приложений. Основное преимущество этого подхода состоит в том, что в приложении можно применять практически тривиальные SQL-операторы. Не нужно выполнять соединения и делать несколько запросов, чтобы получить все необходимые данные. Все необходимое возвращается при выборке всего одной строки.

Четвертый способ использования, создание объектных таблиц по хранимому типу данных, был рассмотрен в главе 6, посвященной таблицам. Поскольку объектные таблицы по "внешним" свойствам аналогичны объектным представлениям (или скорее, наоборот — объектные представления аналогичны объектным таблицам), их использование тоже, фактически, рассмотрено. Я предпочитаю не использовать объектные таблицы. По многим описанным ранее причинам, мне больше нравятся объектные представления реляционных таблиц. Основная причина в том, что, в конечном итоге, почта **всегда** необходимо поддерживать реляционное представление данных, как обеспечивающее различные способы использования данных в приложениях. Объектно-реляционные представления прекрасно подходят для моделирования специфических представлений данных в приложениях.

21

Тщательный контроль доступа

Тщательный контроль доступа (Fine Grained Access Control — FGAC) в Oracle 8i позволяет во время выполнения динамически добавлять условие (конструкцию **WHERE**) ко всем запросам, обращенным к таблице или представлению базы данных. Теперь можно процедурно изменять запрос во время выполнения, другими словами, динамически создавать представление. Можно проверить, кто выполнял запрос, с какого терминала и когда (например, по времени суток) он выполнялся, а затем создать условие на основе этой информации. С помощью контекстов приложений можно безопасно добавлять в среду информацию (например, роль пользователя в отношении приложения) и обращаться к этой информации в процедуре или условии.

В различных публикациях средства FGAC описываются под различными названиями. Обычно используются следующие:

- тщательный контроль доступа (Fine Grained Access Control);
- виртуальная приватная база данных (Virtual Private Database — **VPD**);
- защита на уровне строк (**Row Level Security**), или пакет **DBMS_RLS** (этот PL/SQL-пакет реализует соответствующие возможности).

Чтобы выполнять представленные в этой главе примеры, необходим сервер Oracle версии 8.1.5 или выше. Кроме того, средства тщательного контроля доступа доступны только в редакциях Enterprise и Personal Edition; в Standard Edition эти примеры работать не будут.

В этой главе мы рассмотрим следующее.

- Преимущества использования средств тщательного контроля доступа — простота сопровождения, реализация этих средств на сервере, возможность развития приложений и упрощение их разработки и т.д.

- Два примера в разделе "Как реализованы средства тщательного контроля доступа", демонстрирующие применение *правил защиты* (security policies) и контекстов приложений.
- Проблемы и нюансы, которые необходимо учитывать, в частности особенности функционирования средств тщательного контроля доступа при наличии требований целостности ссылок, кэширование курсоров, особенности экспортирования и импортирования данных и тонкости отладки.
- Ошибки, с которыми можно столкнуться при реализации тщательного контроля доступа в приложениях.

Пример

Предположим, существуют правила защиты, определяющие, какие строки могут просматривать различные группы пользователей. Правила защиты позволяют разработать условие проверки, учитывающее, кто зарегистрирован и какую роль он имеет в системе. Средства тщательного контроля доступа позволяют переписать простой запрос **SELECT * FROM EMP** следующим образом:

Пользователь зарегистрирован как...	***Запрос переписывается так...***	***Комментарии***
Сотрудник	select * from (select * from emp where ename = USER)	Рядовые сотрудники могут просматривать только собственные записи.
Руководитель подразделения	select * from (select * from emp where mgr = (select empno from emp where ename = USER) or ename = USER)	Руководители подразделений могут просматривать свои записи и записи сотрудников своего подразделения,
Сотрудники отдела кадров.	select * from (select * from emp where deptno = SYS_CONTEXT('OurApp', 'ptno'))	Сотрудники отдела кадров могут видеть все записи в данном подразделении. В этом примере представлен способ получения значений переменных из контекста приложения с помощью встроенной функции **SYS_CONTEXT().**

Когда использовать это средство?

В этом разделе рассмотрены причины и варианты использования средств тщательного контроля доступа.

Простота сопровождения

Средства тщательного контроля доступа позволяют с помощью одной таблицы и одной хранимой процедуры справиться с задачей, для решения которой могло бы понадобиться несколько представлений или триггеров, или большой объем специализированной обработки в приложениях.

Подход с использованием нескольких представлений достаточно типичен. Разработчики приложений создают несколько учетных записей в базе данных, например **EMPLOYEE**, **MANAGER**, **HR_REP**, и устанавливают в каждой из соответствующих схем полный набор представлений, выбирающих только необходимые данные. Для рассматриваемого примера в каждой схеме создается отдельное представление **EMP** со специализированным условием выбора данных для соответствующей группы пользователей. Для управления тем, что конечные пользователи могут просматривать, создавать, изменять и удалять, придется создавать до четырех различных представлений для таблицы **EMP** - для операторов **SELECT**, **INSERT**, **UPDATE** и **DELETE**. Это быстро приводит к запредельному увеличению количества объектов базы данных — каждый раз при добавлении новой группы пользователей придется создавать и поддерживать новый набор представлений.

Если правила защиты изменятся (например, если необходимо разрешить руководителям просматривать записи не только непосредственных подчиненных, но и подчиненных следующего уровня), придется пересоздать представление в базе данных, делая недействительными все объекты, которые на него ссылаются. Такой подход приводит не только к увеличению количества представлений в базе данных, но и требует, чтобы пользователи регистрировались от имени нескольких совместно используемых учетных записей, что осложняет контроль работы пользователей. Кроме того, этот подход требует дублирования значительного объема кода в базе данных. При наличии хранимой процедуры, работающей с таблицей **EMP**, придется устанавливать ее в каждой из задействованных схем. Это относится и ко многим другим объектам (триггерам, функциям, пакетам и т.д.). Теперь при внесении изменений в программное обеспечение придется каждый раз изменять N схем, чтобы все пользователи выполняли один и тот же код.

Еще один подход связан с использованием, кроме представлений, триггеров базы данных. Вместо создания отдельных представлений для операторов **SELECT**, **INSERT**, **UPDATE** и **DELETE**, для построчного просмотра выполняемых пользователем изменений используется триггер, принимающий или отвергающий эти изменения. Эта реализация не только приводит к созданию большого количества дополнительных представлений, но и влечет расходы ресурсов на поддержку срабатывания триггера (иногда весьма сложного) для каждой изменяемой строки.

Наконец, можно всю защиту реализовать в приложении, будь то клиентское приложение в случае архитектуры клиент-сервер или сервер приложений. Приложение будет учи-

тывать, кто к нему обращается, и выполнять соответствующий запрос. Приложение по сути реализует собственный механизм тщательного контроля доступа. Серьезным недостатком такого подхода (и вообще любого подхода, использующего для доступа к данным специфические средства приложения) является то, что данные в базе могут использоваться **только** соответствующим приложением. Нельзя использовать никакие средства создания запросов, средства генерации отчетов и т.п., поскольку данные не защищены, если доступ к ним идет не через приложение. Когда защита встроена в приложение, усложняется развитие приложения и добавление новых интерфейсов, т.е. снижается полезность данных.

Средства тщательного контроля доступа позволяют справиться с этими трудностями и избежать потери функциональных возможностей с помощью всего двух объектов - исходной таблицы (или представления) и пакета (или функции) базы данных. Пакет можно изменить в любой момент, разработав новые правила защиты. Вместо поддержки десятков представлений, реализующих правила защиты для объекта, всю соответствующую информацию можно задавать в одном месте.

Контроль доступа выполняется на сервере

С учетом сложности создания и поддержки большого количества представлений разработчики часто реализуют алгоритмы защиты в самом приложении, как было описано выше. Приложение анализирует, кто зарегистрировался и что он запрашивает, а затем отправляет на сервер соответствующий запрос. Это позволяет защитить данные только при доступе к ним через приложение, а вероятность неавторизованного доступа к данным увеличивается, поскольку для этого достаточно подключиться к базе данных с помощью любого инструментального средства, кроме приложения, и выполнить запрос.

При тщательном контроле доступа алгоритмы защиты, определяющие, какие данные может "видеть" пользователь, помещаются в базу данных. При этом гарантируется защита данных, независимо от используемого средства доступа к ним.

Потребность в таких средствах вполне объяснима. В начале и середине 1990-х годов преимущественно использовалась модель клиент-сервер (а еще раньше нормой считалось централизованное выполнение приложений). Большинство клиент-серверных приложений (и практически все централизованные) включали алгоритмы, проверяющие уровень доступа к приложению. Сегодня очень модно использовать серверы приложений и размещать на них все прикладные алгоритмы. По мере переноса клиент-серверных приложений на новую архитектуру разработчики начали переносить алгоритмы защиты с клиентской части и встраивать их в серверы приложений. Это привело к двойной реализации алгоритмов защиты (некоторые клиент-серверные приложения продолжают использоваться), так что теперь поддерживать и отлаживать эти алгоритмы надо в двух местах. Ситуация станет еще хуже при появлении следующей парадигмы программирования. Что случится, когда серверы приложений выйдут из моды? Что делать, если пользователям необходимо инструментальное средство сторонних производителей, обращающееся к данным непосредственно? Если вся защита сосредоточена на промежуточном сервере приложений, это становится невозможным. Если же защита реализуется сервером баз данных, вы готовы к применению любой технологии, как существующей, так и еще не придуманной.

Упрощение разработки приложений

Средства тщательного контроля доступа позволяют отделить алгоритмы защиты от других алгоритмов работы приложения. Разработчик приложения может заняться прикладными алгоритмами, а не алгоритмами безопасного доступа к данным. Поскольку тщательный контроль доступа выполняется полностью на сервере баз данных, эти алгоритмы немедленно наследуются всеми приложениями. Раньше разработчикам приходилось включать алгоритмы защиты в приложения, что усложняло разработку и дальнейшее сопровождение приложений. Если приложения должны обеспечивать защиту доступа к данным, а доступ к одним и тем же данным выполняется во многих компонентах приложения, изменение правил защиты повлияет на десятки модулей приложения. При использовании средств тщательного контроля доступа все соответствующие модули автоматически наследуют новые правила доступа без каких-либо изменений в коде.

Эволюционная разработка приложений

Во многих средах правила защиты изначально строго не сформулированы и со временем могут меняться. При слиянии компаний или ужесточении правил доступа к данным, или появлении новых законов о невмешательстве в частную жизнь правила защиты придется изменять. Если средства контроля доступа реализованы как можно ближе к данным, эти изменения легко учесть при минимальном изменении приложений и инструментальных средств. Новые алгоритмы защиты реализуются в одном месте, а все приложения и средства доступа к базе данных автоматически наследуют новые алгоритмы.

Отказ от совместно используемых учетных записей

При использовании средств тщательного контроля доступа каждый пользователь может регистрироваться от имени уникальной учетной записи. Это позволяет выполнять полный учет и проверку действий на уровне пользователей. В прошлом многие разработчики приложений, столкнувшись с необходимостью обеспечивать разное представление данных для различных пользователей, создавали совместно используемые учетные записи. Например, все сотрудники для доступа к кадровой информации использовали учетную запись **EMPLOYEE**; все руководители — учетную запись **MANAGER** и т.д. Это не позволяло контролировать действия на уровне отдельных пользователей. Невозможно было понять, регистрировался ли пользователь **TKYTE** — в системе работало несколько сеансов от имени учетной записи **EMPLOYEE** (кто бы из сотрудников ни подключался).

При желании можно использовать средства тщательного контроля доступа вместе с такими совместно используемыми учетными записями. Однако эти средства позволяют избежать необходимости создания и использования таких учетных записей.

Поддержка совместно используемых учетных записей

Это дополнение к предыдущему разделу. Средства тщательного контроля доступа не требуют обязательного использования для каждого пользователя отдельной учетной за-

писи; они только позволяют это сделать. С помощью контекста приложения можно использовать средства тщательного контроля доступа и в "однопользовательской" среде, например, используемой при организации пула подключений сервера приложений. Некоторые пулы подключений требуют при регистрации использования одной учетной записи базы данных. Средства тщательного контроля доступа прекрасно работают и в таких средах.

Предоставление доступа к приложению как к службе

Средства тщательного контроля доступа позволяют поставщику прикладных служб (Application Service Provider — ASP), не изменяя существующее приложение, предоставить к нему как к службе доступ множеству клиентов. Предположим, имеется приложение по учету кадров, доступ к средствам которого хотелось бы за деньги предоставлять клиентам по сети Internet. Поскольку клиентов предполагается много и все они требуют гарантий конфиденциальности данных, необходимо придумать способ защиты данных одного клиента от доступа других. Возможны следующие варианты:

- установить, сконфигурировать и поддерживать отдельные экземпляры базы данных для каждого клиента;
- переписать все используемые приложением хранимые процедуры так, чтобы они работали с правами вызывающего (это будет описано в главе 23), и создать для каждого клиента отдельную схему;
- использовать один экземпляр базы данных и одну схему со средствами тщательного контроля доступа.

Первый вариант крайне нежелателен. Неизбежные расходы ресурсов на поддержку отдельного экземпляра базы данных для каждого клиента, имеющего всего десяток сотрудников, не позволяют реально его использовать. Для крупных клиентов с сотнями или тысячами пользователей он вполне оправдан. Для массы же мелких клиентов, каждый из которых добавляет пять-шесть конечных пользователей, создавать отдельную базу данных слишком расточительно.

Второй вариант потенциально требует переписать приложение. Целью является создание для каждого клиента отдельной схемы со своим набором таблиц. Любую хранимую процедуру надо создавать так, чтобы она работала с таблицами, доступными только для текущего зарегистрированного пользователя (клиента). Обычно хранимым процедурам доступны те же объекты, что и **создателю** процедуры — надо только убедиться, что используются подпрограммы, работающие с правами вызывающего и что в приложении нигде явно не указаны схемы. Например, нельзя использовать оператор **SELECT * FROM SCOTT.EMP** - только SELECT * **FROM EMP.** Это касается не только PL/SQL-процедур. Для любого внешнего кода на языках Java или Visual Basic тоже должны соблюдаться эти правила (и не использоваться имена схем). Поэтому второй вариант также нежелателен, да еще и связан с необходимостью поддержки многих сотен пользовательских схем.

Третий вариант — использование средств тщательного контроля доступа — наиболее безболезненный и простой. Можно, например, добавить в каждую таблицу, требующую защиты, столбец с идентификатором организации. Для поддержки значений в этом столбце надо использовать триггер (чтобы не пришлось изменять приложение). Триггер будет брать соответствующее значение из контекста приложения, устанавливаемого в системном триггере **ON LOGON.** Правила защиты будут задавать условие выбора строк только соответствующей организации. При этом можно ограничивать доступ к данным не только по идентификатору клиента, но и по любым другим необходимым условиям. В нашей системе кадрового учета добавлять можно не только условие **WHERE COMPANY = ЗНАЧЕНИЕ,** но и дополнительные условия, в зависимости от того, работает ли с системой рядовой сотрудник, руководитель или сотрудник одела кадров. Можно пойти еще дальше и добавить условия фрагментации, чтобы физически отделить данные крупных клиентов с целью обеспечения надежности хранения и высокой доступности.

Как реализованы средства тщательного контроля доступа

Средства тщательного контроля доступа в Oracle 8i реализуются с помощью двух конструкций.

- **Контекст приложения.** Это пространство имен с соответствующим набором пар атрибут/значение. Например, в контексте **OurApp** можно обращаться к переменным **DeptNo, Mgr** и т.д. Контекст приложения всегда привязан к PL/SQL-пакету. Этот пакет — единственный метод установки значений в контексте. Чтобы установить значение атрибута **DeptNo** в нашем контексте **OurApp,** необходимо обратиться к соответствующему пакету, связанному с контекстом **OurApp.** Этот пакет может корректно устанавливать значения в контексте **OurApp** (вы сами его написали, поэтому и считается, что он будет правильно устанавливать значения в контексте). Это предотвращает установку значений в контексте приложений злонамеренными пользователями с целью получения несанкционированного доступа к информации. Читать значения в контексте приложения может кто угодно, но устанавливать их может только один пакет.

- **Правила защиты.** Это созданная разработчиком функция, возвращающая условие для динамического фильтрования данных при выполнении запроса. Эту функцию можно привязывать к определенной таблице или представлению, и вызываться она может для всех или только для некоторых операторов, обращающихся к таблице. Это означает, что можно задать одни правила для оператора **SELECT,** другие — для **INSERT** и третьи — для операторов **UPDATE** и **DELETE.** Обычно в этой функции используются значения атрибутов в контексте приложений для создания соответствующего условия (например, проверяется, кто зарегистрировался и что он пытается сделать и создается соответствующее подмножество строк для работы). Следует помнить, что для пользователя **SYS** (или **INTERNAL)** правила защиты никогда не применяются (соответствующие функции просто никогда не вызываются); эти пользователи всегда могут читать и изменять все данные.

Также следует упомянуть средства сервера Oracle 8i, расширяющие возможности средств тщательного контроля доступа.

- **Функция SYS_CONTEXT.** Эта функция используется в языках SQL и PL/SQL для доступа к значениям в контексте приложения. Подробное описание этой функции и список стандартных значений в автоматически устанавливаемом сервером Oracle контексте **USERENV** можно найти в руководстве *Oracle SQL Reference.* Из этого контекста можно получить имя пользователя, организовавшего сеанс, IP-адрес клиента и другую полезную информацию.
- **Триггеры на событие регистрации в базе данных.** Они позволяют выполнять любой код при регистрации пользователя в базе данных. Это очень удобно для настройки первоначального, стандартного контекста приложения.
- **Пакет DBMS_RLS.** Этот пакет обеспечивает функциональный интерфейс для добавления, удаления, изменения, включения и отключения правил защиты. Соответствующие подпрограммы можно вызвать из любого языка программирования или среды, из которых можно подключиться к СУБД Oracle.

Чтобы использовать средства тщательного контроля доступа, разработчику, помимо стандартных ролей **CONNECT** и **RESOURCE** (или соответствующих им привилегий), необходимы следующие привилегии.

- **EXECUTE_CATALOG_ROLE.** Эта роль позволяет разработчику выполнять подпрограммы пакета **DBMS_RLS.** Достаточно также, подключившись как **SYS,** предоставить пользователю привилегию на выполнение пакета **DBMS_RLS.**
- **CREATE ANY CONTEXT.** Эта привилегия позволяет разработчику создавать контексты приложений.

Контекст приложения создается с помощью SQL-оператора:

```
SQL> create or replace context OurApp using Our_Context_Pkg;
```

Здесь **OurApp** — имя контекста, а **Our_Context_Pkg** — PL/SQL-пакет, которому разрешается устанавливать значения атрибутов контекста. При реализации средств тщательного контроля доступа контексты приложений существенны по двум причинам.

- **Они обеспечивают надежный способ установки значений переменных в пространстве имен.** Устанавливать значения в контексте можно только с помощью PL/SQL-пакета, связанного с этим контекстом. Это гарантирует целостность значений в контексте. Поскольку контекст используется для ограничения или обеспечения доступа к данным, необходимо гарантировать целостность значений в контексте.
- **Ссылки на значения атрибутов контекста** в **SQL-запросе обрабатываются как связываемые переменные.** Например, если установить значение атрибута DeptNo в контексте **OurApp** и реализовать правило защиты, возвращающее конструкцию **WHERE deptno = SYS_CONTEXT('OurApp','DeptNo'),** соответствующий оператор не будет в разделяемом пуле уникальным, поскольку обращение к функции **SYS_CONTEXT** аналогично **deptno = :bl. B** сеансах можно будет задавать различные значения атрибута **Deptno,** но все они будут повторно использовать одни и те же оптимизированные планы запросов.

Пример 1: Реализация правил защиты

Чтобы продемонстрировать возможности средств тщательного контроля доступа, будут реализованы очень простые правила зашиты. Зададим следующие правила.

- Если текущий пользователь является **владельцем** таблицы **(OWNER),** он может обращаться ко всем ее строкам.
- В противном случае пользователь может обращаться только к строкам, у которых значение в столбце **OWNER** совпадает с его именем.
- Кроме того, добавлять можно только строки, в столбце **OWNER** которых указано имя обращающегося пользователя. Попытки добавить строку с другим значением в этом столбце отвергаются.

Для реализации этих правил необходимо создать следующую PL/SQL-функцию:

```
tkyte@TKYTE816> create or replace
  2  function security_policy_function(p_schema in varchar2,
  3                p_object in varchar2)
  4  return varchar2
  5  as
  6  begin
  7      if (user = p_schema) then
  8          return '';
  9      else
 10          return 'owner = USER';
 11      end if;
 12  end;
 13  /
Function created.
```

Этот пример показывает общую структуру функции, реализующей правила защиты. Эта функция всегда возвращает значение типа **VARCHAR2.** Возвращаемое значение представляет собой условие, которое будет добавлено к запросу. Фактически это условие будет добавляться к таблице или представлению, к которым применяется это правило защиты, с помощью подставляемого представления (inline view):

```
Запрос:          SELECT        * FROM T
Будет переписан как: SELECT * FROM (SELECT * FROM T WHERE owner = USER)
или:                 SELECT * FROM (SELECT * FROM T)
```

Кроме того, все функции, реализующие правила защиты, должны принимать два параметра в режиме IN: имя схемы, которой **принадлежит** объект, и имя объекта, к которому применяется функция. Их значения можно при необходимости использовать в функции, реализующей правила защиты.

Итак, в нашем примере условие **owner = USER** будет динамически добавляться ко всем запросам к таблице, с которой связана эта функция, эффективно ограничивая множество строк, доступных пользователю. Пустое условие будет возвращаться только в том случае, если текущий зарегистрированный пользователь является владельцем таблицы. Вернуть пустое условие — то же самое, что вернуть условие **1=1** или **True.** Воз-

врат значения Null равносилен возвращению пустого условия. В представленном выше примере вместо пустой строки можно было с тем же результатом возвращать Null.

Чтобы связать функцию с таблицей, используется рассматриваемая далее PL/SQL-процедура **DBMS_RLS.ADD_POLICY**. В нашем примере имеется следующая таблица, а сеанс выполняется от имени пользователя **TKYTE**:

```
tkyte@TKYTE816> create table data_table
  2  (some_data    varchar2(60),
  3   OWNER        varchar2(30) default USER
  4  )
  5  /

Table created.

tkyte@TKYTE816> grant all on data_table to public-

Grant succeeded.

tkyte@TKYTE816> create public synonym data_table for data_table;

Synonym created.

tkyte0TKYTE816> insert into data_table (some_data) values ('Некие данные');

1 row created.

tkyte@TKYTE816> insert into data_table (some_data, owner)
  2  values ('Некие данные, принадлежащие пользователю SCOTT', 'SCOTT');

1 row created.

tkyte0TKYTE816> commit;

Commit complete.

tkyte0TKYTE816> select * from data_table;

SOME_DATA                                          OWNER
Некие данные                                       TKYTE
Данные, принадлежащие пользователю SCOTT           SCOTT
```

Теперь привяжем написанную ранее функцию защиты к этой таблице с помощью следующего обращения к пакету **DBMS_RLS**:

```
tkyte@TKYTE816> begin
  2     dbms_rls.add_policy
  3     (object_schema   => 'TKYTE',
  4      object_name     => 'data_table',
  5      policy_name     => 'MY_POLICY',
  6      function_schema => 'TKYTE',
  7      policy_function => 'security_policy_function',
  8     statement_types => 'select, insert, update, delete',
  9      update_check    => TRUE,
 10      enable          => TRUE
 11     );
 12  end;
 13  /

PL/SQL procedure successfully completed.
```

Процедура **ADD_POLICY** — одна из ключевых процедур пакета **DBMS_RLS**. Именно она позволяет добавить правило зашиты для таблицы. Мы передали процедуре следующие параметры.

- **OBJECT_SCHEMA**. Имя владельца таблицы или представления. Если оставить стандартное значение **Null**, оно будет интерпретироваться как имя текущего зарегистрированного пользователя. Для полноты рассмотренного выше примера я передал имя пользователя.
- **OBJECT_NAME**. Имя таблицы или представления, для которого добавляется правило.
- **POLICY_NAME**. Уникальное имя для этого правила. Это имя используется в дальнейшем для включения, отключения, изменения или удаления правила.
- **FUNCTION_SCHEMA**. Имя владельца функции, возвращающей условие. Оно обрабатывается аналогично параметру **OBJECT_SCHEMA**. Если оставлено стандартное значение **Null**, используется имя текущего зарегистрированного пользователя.
- **POLICY_FUNCTION**. Имя функции, возвращающей условие.
- **STATEMENT_TYPES**. Список типов операторов, к которым применяется правило. Может представлять собой любое сочетание **INSERT**, **UPDATE**, **SELECT** и **DELETE**, перечисленных через запятую. Стандартное значение — все четыре оператора. Для наглядности я задал список явно.
- **UPDATE_CHECK**. Этот параметр влияет только на обработку операторов **INSERT** и **UPDATE**. Если параметр имеет значение True (стандартное значение — **False**), будет выполняться проверка, доступны ли вставленные или измененные данные текущему пользователю в соответствии с заданным условием. Другими словами, если задано значение True, нельзя вставить данные, которые нельзя будет выбрать из таблицы в соответствии с возвращаемым функцией условием.
- **ENABLE**. Задает, включено это правило или нет. Стандартное значение — True.

Теперь, после вызова процедуры **ADD_POLICY**, ко всем операторам **ЯМД**, применяемым к таблице **DATA_TABLE**, будет добавляться условие, возвращаемое функцией **SECURITY_POLICY_FUNCTION**, независимо от того, из какой среды поступил оператор ЯМД. Другими словами, независимо от приложения, обращающегося к данным. Чтобы увидеть это в действии, выполним:

```
tkyte@TKYTE816>connect system/manager
system@TKYTE816>select * from data_table;
no rows selected

system@TKYTE816>connect scott/tiger
scott@TKYTE816>select * from data_table;

SOME_DATA                                OWNER

Данные, принадлежащие пользователю SCOTT  SCOTT
```

Итак, этот пример показывает, что строки фильтруются — пользователь **SYSTEM** не получает из этой таблицы никаких данных. Причина в том, что условие **WHERE OWNER = USER** не выполняется ни для одной из существующих строк данных. При регистрации от имени пользователя **SCOTT**, однако, можно получить единственную строку, принадлежащую пользователю **SCOTT**. Продолжим пример и попытаемся применить к таблице ряд операторов ЯМД:

```
sys@TKYTE816>connect scott/tiger

scott@TKYTE816>insert into data_table (some_data)
  2 values ('Новые данные');

1 row created.

scott@TKYTE816>insert into data_table (some_data, owner)
  2 values ('Новые данные, принадлежащие пользователю SYS', 'SYS')
  3 /
insert into data_table ( some_data, owner )
            *
ERROR at line 1:
ORA-28115: policy with check option violation
```

```
scott@TKYTE816> select * from data_table;

SOME_DATA                                          OWNER

Данные, принадлежащие пользователю SCOTT           SCOTT
Новые данные                                       SCOTT
```

Можно создавать данные, которые будут доступны, но если они недоступны, возвращается сообщение об ошибке **ORA-28115**, поскольку при добавлении правила в вызове процедуры **dbms_rls.add_policy** было передано значение:

```
9       update_check     => TRUE);
```

Это аналогично созданию представления с конструкцией **CHECK OPTION**. Разрешается создавать только те данные, которые можно потом выбрать. По умолчанию можно создавать данные, не выбираемые в соответствии с правилами защиты.

Теперь, в соответствии с реализованным правилом защиты, владелец таблицы видит все строки и имеет возможность создавать любую строку. Чтобы убедиться в этом, регистрируемся как пользователь **TKYTE** и пытаемся выполнить следующие действия:

```
scott@TKYTE816>connect tkyte/tkyte
tkyte@TKYTE816>insert into data_table (some_data, owner)
  2 values ('Новые данные, принадлежащие пользователю SYS', 'SYS')
  3 /

1 row created.

tkyte@TKYTE816>select * from data_table
  2 /
```

```
SOME_DATA                                          OWNER

Некие данные                                       TKYTE
Данные, принадлежащие пользователю SCOTT           SCOTT
Новые данные                                       SCOTT
Новые данные, принадлежащие пользователю SYS       SYS
```

Итак, пример показывает, что на пользователя **TKYTE** правило защиты не распространяется. Интересно отметить, что в случае регистрации от имени пользователя **SYS** наблюдается следующая особенность:

```
tkyte@TKYTE816>  connect sys/change_on_install
Connected.
sys@TKYTE816> select * from data_table;
SOME_DATA                                          OWNER

Некие данные                                       TKYTE
Данные, принадлежащие пользователю SCOTT           SCOTT
Новые данные                                       SCOTT
Новые данные, принадлежащие пользователю SYS       SYS
```

Правила зашиты не применяются для специального пользователя **SYS** (а также при регистрации от имени **INTERNAL** или как **SYSDBA).** Это сделано специально. Учетные записи с привилегиями **SYSDBA** предназначены для решения задач администрирования и соответствующим пользователям доступны все данные. Это особенно важно учитывать при экспортировании данных. Если только экспортирование не выполняется с привилегиями **SYSDBA,** применяются правила защиты. При использовании учетной записи без привилегий **SYSDBA** и обычном экспорте вы получите не **все** данные!

Пример 2: Использование контекстов приложений

В этом примере мы реализуем правила защиты информации о сотрудниках (для отдела кадров крупной компании). Будем использовать простые таблицы **EMP** и **DEPT,** принадлежащие пользователю **SCOTT,** и добавим таблицу с информацией о сотрудниках, которые отвечают за кадровые вопросы в том или ином отделе. При этом необходимо реализовать следующие правила защиты.

Руководитель отдела может:

- просматривать свою собственную запись, а также записи для всех своих подчиненных;
- изменять записи своих непосредственных подчиненных.

Рядовой сотрудник может:

- просматривать свою собственную запись.

Ответственный за кадровые вопросы в отделе может:

- читать все записи для отдела (предполагается, что ответственный за кадровые вопросы в определенный момент времени работает только в одном отделе);

- изменять все записи для отдела;
- вставлять записи для соответствующего отдела;
- удалять записи для соответствующего отдела.

Как было сказано, приложение будет использовать копии существующих таблиц **EMP** и **DEPT** из схемы пользователя **SCOTT** и дополнительную таблицу, **HR_REPS,** позволяющую назначить ответственного за кадровые вопросы в отделе. При регистрации желательно автоматически назначать пользователю соответствующую роль в приложении. Так что, например, при регистрации ответственного за кадровые вопросы он сразу бы получал соответствующую роль в приложении.

Для начала необходимо создать в базе данных ряд учетных записей. Эти учетные записи создаются для владельца приложения и пользователей. В данном случае владелец приложения — пользователь **TKYTE**, и соответствующая схема будет содержать копии таблиц **EMP** и **DEPT** из демонстрационной схемы **SCOTT**. Пользователи именуются так же, как сотрудники в таблице **EMP** (**KING**, **BLAKE** и т.д.). Для создания и настройки всех этих учетных записей использовался следующий сценарий. Сначала удаляем и пересоздаем пользователя TKYTE, предоставляя ему роли **CONNECT** и **RESOURCE**:

```
sys@TKYTE816>  drop user tkyte cascade;
User dropped.
sys@TKXTE816> create user tkyte identified by tkyte
  2  default tablespace data
  3  temporary tablespace temp;
User created.
sys@TKYTE816>grant connect, resource to tkyte;
Grant succeeded.
```

Теперь предоставим пользователю минимальные привилегии, необходимые для организации тщательного контроля доступа. Вместо привилегии **EXECUTE ON DBMS_RLS** можно предоставить роль **EXECUTE_CATALOG**:

```
sys@TKYTE816>grant execute on dbms_rls to tkyte;
Grant succeeded.
sys@TKYTE816>grant create any context to tkyte;
Grant succeeded.
```

Следующая привилегия необходима для создания триггера на событие регистрации в базе данных, который надо будет создать в дальнейшем:

```
sys@TKYTE816> grant administer database trigger to tkyte;
Grant succeeded.
```

Теперь создадим учетные записи сотрудников и руководителей для пользователей приложения. Для каждого сотрудника в таблице **EMP** (кроме **SCOTT**) будет создана учетная запись, имя которой совпадает со значением в столбце **ENAME**. Во многих базах данных учетная запись **SCOTT** уже существует:

```
sys@TKYTE816> begin
  2      for x in (select ename
  3             from scott.emp  where ename <> 'SCOTT')
  4      loop
  5         execute immediate 'grant connect to ' || x.ename ||
  6                 ' identified by ' || x.ename;
  7      end loop;
  8  end;
  9  /

PL/SQL procedure successfully completed.

sys@TKYTE816> connect scott/tiger

scott@TKYTE816> grant select on emp to tkyte;

Grant succeeded.

scott@TKYTE816> grant select on dept to tkyte;

Grant succeeded.
```

Приложением используется следующая простая схема. Сначала копируются таблицы EMP и DEPT из схемы пользователя SCOTT. Для этих таблиц мы также добавим декларативные требования целостности ссылок:

```
scott@TKYTE816> connect tkyte/tkyte

tkyte@TKYTE816> create table dept as select * from scott.dept;

Table created.

tkyte@TKYTE816> alter table dept add constraint dept_pk primary
key(deptno);

Table altered.

tkyte@TKYTE816> create table emp_base_table as select * from scott.emp;

Table created.

tkyte@TKYTE816> alter table emp_base_table add constraint
  2 emp_pk primary key(empno);

Table altered.

tkyte@TKYTE816> alter table emp_base_table add constraint emp_fk_to_dept
  2  foreign key (deptno) references dept(deptno);

Table altered.
```

Теперь добавим несколько индексов и требований. Создадим индексы, которые будут использоваться функциями контекста приложения для повышения их производительности. Мы должны иметь возможность быстро определять, является ли данный пользователь руководителем отдела:

```
tkyte@TKYTE816> create index emp_mgr_deptno_idx on emp_base_table(mgr);
Index created.
```

Необходимо также быстро получать по имени пользователя значение EMPNO и обеспечить уникальность имен пользователей в приложении:

```
tkyte@TKYTE816>  alter table emp_base_table
  2  add constraint
  3  emp_ename_unique unique(ename);

Table altered.
```

Теперь создадим представление **EMP** для таблицы **EMP_BASE_TABLE.** Правила защиты будут задаваться для этого представления, а приложение будет обращаться к нему для получения, вставки, изменения и удаления данных. Зачем используется представление, я объясню чуть позже:

```
tkyte@TKYTE816>create view emp as select * from emp_base_table;

View created.
```

Теперь создадим таблицу для хранения информации о сотрудниках, ответственных за кадровые вопросы в отделах. Для этого используем таблицу, организованную по индексу. Поскольку к таблице будет выполняться единственный запрос, SELECT * FROM HR_REPS WHERE USERNAME = :X AND DEPTNO = :Y, таблица традиционной структуры нам не нужна:

```
tkyte@TKYTE816>create table hr_reps
  2  (username    varchar2(30),
  3     deptno      number,
  4      primary key(username,deptno)
  5  )
  6 organization index;

Table created.
```

Теперь зададим ответственных за кадровые вопросы в отделах:

```
tkyte@TKYTE816>insert into hr_reps values ('KING', 10);

1 row created.

tkyte@TKYTE816>insert into hr_reps values( 'KING', 20);

1 row created.

tkyte@TKYTE816>insert into hr_reps values ('KING', 30);

1 row created.

tkyte@TKYTE816>insert into hr_reps values ('BLAKE', 10);

1 row created.

tkyte@TKYTE816>insert into hr_reps values ('BLAKE', 20);

1 row created.

tkyte@TKYTE816> commit;

Commit complete.
```

Теперь, когда таблицы приложения, **EMP**, **DEPT** и **HR_REPS**, созданы, создадим процедуру, которая позволит устанавливать контекст приложения. Он будет содержать три атрибута: номер **(EMPNO)** текущего зарегистрированного пользователя, имя пользователя (**USERNAME**) и роль пользователя в приложении **(EMP, MGR** или **HR_REP).** При создании конструкции **WHERE** для выполняемых пользователем операторов фун-

кция, динамически формирующая условие, будет использовать роль, хранящуюся в контексте приложения.

Для определения роли используется информация из таблиц **EMP_BASE_TABLE** и HR_REPS. Вот вам и ответ на вопрос, зачем создавать таблицу EMP_BASE_TABLE и отдельное представление **EMP** как **SELECT * FROM EMP_BASE_TABLE**. Для этого имеются две причины:

- данные в таблице сотрудников используются для реализации правил защиты;
- данные из этой таблицы считываются при попытке задать контекст приложения.

Для чтения данных о сотрудниках необходимо настроить контекст приложения, но для настройки контекста приложения необходимо получить данные о сотрудниках. Это проблема определения, что первично: курица или яйцо? Мы создадим представление (EMP), к которому будут обращаться все приложения, и обеспечим защиту этого представления. Исходная таблица **EMP_BASE_TABLE** будет использоваться функцией, реализующей правила защиты. По таблице **EMP_BASE_TABLE** можно определить, кто является руководителем отдела и кто у него в подчинении. Приложения и пользователи никогда не будут обращаться к таблице **EMP_BASE_TABLE** — только функция, реализующая правила защиты. Для этого мы не будем предоставлять другим пользователям прав доступа к базовой таблице; при этом невозможность работы с ней обеспечит сервер.

В этом примере мы выбрали автоматическую установку контекста приложения при регистрации. Это стандартная процедура, если ее можно использовать, для автоматической настройки контекста приложения. Иногда этого недостаточно. Если при регистрации не хватает информации для определения того, каким должен быть контекст, придется установить его атрибуты вручную с помощью вызова соответствующей процедуры. Это часто приходится делать при использовании сервера приложений, подключающего всех пользователей к базе данных через одну совместно используемую учетную запись. Сервер приложений вызовет процедуру базы данных, передав ей имя "реального" пользователя для правильной настройки контекста.

Ниже представлена написанная нами процедура для установки значений атрибутов контекста. Мы знаем все особенности ее работы, поэтому можем ей доверять. Она позволяет реализовать необходимые правила защиты, устанавливая в контексте только соответствующее имя пользователя, название роли и номер сотрудника. В дальнейшем при обращении к этим значениям можно быть уверенными в том, что они заданы правильно и безопасно. Процедура будет автоматически выполняться триггером **ON LOGON**. В такой реализации процедура позволяет поддерживать 3-уровневые приложения, использующие пул подключений и всего одну учетную запись в базе данных. Необходимо предоставить право на выполнение этой процедуры учетной записи, используемой пулом подключений, а затем на сервере приложений выполнять эту процедуру, причем не используя стандартное значение — имя текущего пользователя, подключившегося к базе данных, а передавая имя пользователя как параметр.

```
tkyte@TKYTE816> create or replace
  2  procedure set_app_role(p_username in varchar2
  3                         default sys_context('userenv','session_user'))
  4  as
```

```
 5      l_empno    number;
 6      l_cnt      number;
 7      l_ctx      varchar2(255) default 'Hr_App_Ctx';
 8  begin
 9      dbms_session.set_context(l_ctx, 'UserName', p_username);
10      begin
11         select empno into l_empno
12           from emp_base_table
13          where ename = p_username;
14         dbms_session.set_context(l_ctx, 'Empno', l_empno);
15      exception
16         When NO_DATA_FOUND then
17            — Пользователя нет в таблице emp - это, должно быть,
-> ответственный за кадры.
18            NULL;
19      end;
20
21
22      — Сначала посмотрим, не является ли этот пользователь
-> ответственным за кадры; если нет -
23      — вдруг это руководитель; в противном случае устанавливаем
-> пользователю роль EMP.
24
25     select count(*) into l_cnt
26       from dual
27      where exists
28       (select NULL
29          from hr_reps
30         where username = p_username
31       );
32
33      if (l_cnt o 0)
34     then
35        dbms_session.set_context(l_ctx, 'RoleName', 'HR_REP');
36         else
37         — Проверим, не является ли пользователь руководителем.
38         — Если нет, он получает роль EMP.
39
40         select count(*) into l_cnt
41           from dual
42          where exists
43           (select NULL
44               from emp_base_table
45              where mgr = to_number(sys_context(l_ctx,'Empno'))
46           );
47          if (l_cnt o 0)
48         then
49            dbms_session.set_context(l_ctx, 'RoleName', 'MGR');
50         else
51         — Роль EMP может получить каждый.
52           dbms_session.set_context(l_ctx, 'RoleName', 'EMP');
53         end if;
```

```
54        end if;
55  end;
56  /

Procedure created.
```

Создадим контекст приложения. Контекст будет иметь имя **HR_APP_CTX** (совпадающее с именем предыдущей процедуры). Создавая контекст, обратите внимание, как он привязывается к только что созданной процедуре, — только она сможет устанавливать значения атрибутов в этом контексте:

```
tkyte@TKYTE816>  create or replace context Hr_App_Ctx using SET_APP_ROLE
  2  /

Context created.
```

Чтобы автоматизировать настройку контекста, используем триггер базы данных на событие регистрации, в котором будет вызываться процедура, устанавливающая значения контекста:

```
tkyte@TKYTE816>  create or replace trigger APP_LOGON_TRIGGER
  2  after logon on database
  3  begin
  4          set_app_role;
  5  end;
  6  /

Trigger created.
```

Итак, мы создали процедуру, задающую роль для текущего зарегистрированного пользователя. Эта процедура будет вызываться не более одного раза за сеанс, гарантируя, что атрибут **RoleName** однократно получает значение при регистрации, которое затем не изменяется. Поскольку в зависимости от значения **RoleName** функция, реализующая правила защиты, будет возвращать различные значения, в версиях Oracle 8.1.6 и 8.1.6 нельзя разрешать пользователям изменять свою роль после установки. В противном случае возникнет потенциальная проблема с кэшированными курсорами и "старыми" условиями (см. описание соответствующей проблемы в разделе "Проблемы" далее в этой главе; в версии 8.1.7 проблема в основном решена). Кроме того, мы находим значение **EMPNO** для текущего пользователя. Это дает нам два преимущества.

- Возможность проверить, является ли пользователь сотрудником. Получение сообщения об ошибке **NO_DATA_FOUND** позволяет судить, что подключившийся пользователь не является сотрудником. Поскольку его атрибут **EMPNO** не получает значения, этот пользователь не увидит данных, если только не является ответственным за кадры.
- Часто используемое значение помещается в контекст приложения. Теперь можно быстро обратиться к таблице **EMP** по значению **EMPNO** для текущего пользователя, что мы и будем делать далее в функции, формирующей условия.

Затем мы создали объект (контекст приложения) и связали его с созданной ранее процедурой **SET_APP_ROLE.** В результате **только** эта процедура может устанавливать значения в данном контексте. Вот почему контекст приложения можно безопасно использовать и доверять получаемым результатам. Мы точно знаем, какой фрагмент кода может

устанавливать значения в контексте, и мы уверены, что они устанавливаются правильно (ведь мы же сами написали эту процедуру). Следующий пример показывает, что произойдет при попытке установить значения в контексте из другой процедуры:

```
tkyte@TKYTE816> begin
  2    dbms_session.set_context('Hr_App_ctx',
  3                             'RoleName',  'MGR');
  4  end;
  5  /
begin
*
ERROR at line 1:
ORA-01031: insufficient privileges
ORA-06512: at "SYS.DBMS_SESSION", line 58
ORA-06512: at line 2
```

Чтобы проверить логику работы процедуры, попытаемся использовать ее от имени различных пользователей и посмотрим, какие роли мы можем устанавливать и какие значения устанавливаются в контексте. Начнем с пользователя **SMITH**. Это рядовой сотрудник. Он никем не руководит и не отвечает за кадры. Для получения значений, установленных в контексте, воспользуемся общедоступным представлением SESSION_CONTEXT:

```
tkyte@TKYTE816>connect smith/smith

smith@TKYTE816>column namespace format a10
smith@TKYTE816>column attribute format a10
smith@TKYTE816>column value    format a10
smith@TKYTE816>select * from session_context;

NAMESPACE  ATTRIBUTE  VALUE

HR_APP_CTX ROLENAME   EMP
HR_APP_CTX USERNAME   SMITH
HR_APP_CTX EMPNO      7369
```

Как видите, все работает, как и предполагалось. Пользователь SMITH успешно получил соответствующие значения атрибутов **USERNAME, EMPNO и ROLENAME в** контексте **HR_APP_CTX.**

Подключившись от имени другого пользователя, мы видим, как работает процедура, попутно используя другой способ проверки значений в контексте приложения:

```
smith@TKYTE816>    connect blake/blake
blake@TKYTE816> declare
  2     l_AppCtx    dbms_session.AppCtxTabTyp;
  3     l_size       number;
  4  begin
  5     dbms_session.list_context(l_AppCtx, l_size);
  6     for i in 1 .. l_size loop
  7        dbms_output.put(l_AppCtx(i).namespace || '.');
  8     dbms_output.put(l_AppCtx(i).attribute      || ' = ');
  9        dbms_output.put_line(l_AppCtx(i).value);
 10     end loop;
 11  end;
```

```
  12  /
HR_APP_CTX.ROLENAME = HR_REP
HR_APP_CTX.USERNAME = BLAKE
HR_APP_CTX.EMPNO = 7698

PL/SQL procedure successfully completed.
```

На этот раз мы зарегистрировались как пользователь **BLAKE**, который является руководителем отдела 30 и ответственным за кадры в отделах 10 и 30. После регистрации видно, что контекст установлен правильно: для пользователя установлена роль HR_REP, имя пользователя и номер сотрудника. При этом показано, как получить пары атрибут/значение из контекста сеанса с помощью процедуры **DMBS_SESSION.LIST_CONTEXT**. Этот пакет общедоступен, поэтому все пользователи смогут проверять значения атрибутов своего контекста с помощью этого метода, в дополнение к рассмотренному ранее представлению **SESSION_CONTEXT**.

Убедившись, что контекст сеанса устанавливается так, как предполагалось, можно переходить к созданию функций, реализующих правила защиты. Эти функции будут вызываться сервером во время выполнения для динамического добавления условия к операторам. Такие динамически формируемые условия ограничивают множество данных, которые пользователь может читать или записывать. Мы создадим отдельные функции для операторов **SELECT**, операторов **UPDATE** и операторов **INSERT/DELETE**. Дело в том, что каждый из этих операторов может обращаться к различным подмножествам строк. Выбирать можно больше данных, чем изменять (например, сотрудник может просматривать свою запись, но не может ее менять). Только специально назначенные пользователи могут вставлять и удалять строки, поэтому условия для этих операторов тоже отличаются:

```
blake@TKYTE816> connect tkyte/tkyte
tkyte@TKYTE816> create or replace package hr_predicate_pkg
  2  as
  3      function select_function(p_schema in varchar2,
  4                               p_object in varchar2) return varchar2;
  5
  6      function update_function(p_schema in varchar2,
  7                               p_object in varchar2) return varchar2;
  8
  9      function insert_delete_function(p_schema in varchar2,
 10                                      p_object in varchar2) return varchar2;
 11  end;
 12  /
Package created.
```

Ниже представлена реализация пакета **HR_PREDICATE_PKG**. Начнем с глобальных переменных:

```
tkyte@TKYTE816>  create or replace package body hr_predicate_pkg
  2  as
  3
  4  g_app_ctx constant varchar2(30) default 'Hr_App_Ctx';
  5
```

```
6  g_sel_pred     varchar2(1024) default NULL;
7  g_upd_pred     varchar2(1024) default NULL;
8  g_ins_del_pred varchar2(1024) default NULL;
9
```

Константа **G_APP_CTX** содержит имя контекста приложения. Если когда-либо потребуется переименовать контекст, можно изменить значение этой, используемой в остальномкоде константы. Если придется это имя менять, достаточно будет поменять значение константы в одном месте и перекомпилировать тело пакета. Остальные три глобальные переменные будут содержать условия. Этот пример создавался для версии Oracle 8.1.6. В этой версии есть проблема, связанная с кэшированием курсоров и средствами тщательного контроля доступа (подробнее см. в разделе "Проблемы" далее). Начиная с версии Oracle 8.1.7, этот прием программирования использовать необязательно. В данном случае реализуется правило, запрещающее менять роль **после** регистрации. Однажды сгенерированные в сеансе условия возвращаются для всех запросов. Мы не будем заново генерировать их для каждого запроса, так что любые изменения роли не повлияют на результат, пока пользователь не завершит сеанс и не зарегистрируется снова (или не сбросит состояние сеанса с помощью вызова **DBMS_SESSION.RESET_PACKAGE**).

Теперь перейдем к первой из генерирующих условие функций. Она генерирует условие для операторов **SELECT**, которые обращаются к представлению **EMP.** Обратите внимание, что она просто устанавливает значение глобальной переменной пакета **G_SEL_PRED** (Global SELect PREDicate — глобальное условие для **SELECT**) в зависимости от значения атрибута контекста **RoleName.** Если атрибут контекста не установлен, функция возбуждает исключительную ситуацию, которая приводит к неудачному завершению запроса:

```
10
11 function select_function(p_schema in varchar2,
12                          p_object in varchar2) return varchar2
13 is
14 begin
15
16      if (g_sel_pred is NULL)
17      then
18          if (sys_context(g_app_ctx, 'RoleName') = 'EMP')
19          then
20             g_sel_pred:=
21                'empno=sys_context (''' ||g_app_ctx||''','EmpNo')';
22          elsif (sys_context(g_app_ctx, 'RoleName') = 'MGR')
23          then
24             g_sel_pred :=
25              'empno in (select empno
26                    from emp_base_table
27                   start with empno =
28                          sys_context('''||g_app_ctx||''','EmpNo')
29                connect by prior empno = mgr)';
30
31           elsif (sys_context(g_app_ctx, 'RoleName') = 'HR_REP')
```

```
32          then
33             g_sel_pred := 'deptno in
34                       (select deptno
35                          from hr_reps
36                         where username =
37                    sys_context(''''||g_app_ctx||''','',''UserName''))';
38
39          else
40            raise_application_error(-20005, 'Роль не установлена');
41         end if;
42      end if;
43
44       return g_sel_pred;
45 end;
46
```

Теперь перейдем к функции, возвращающей условие для операторов изменения. Алгоритм ее очень похож на алгоритм предыдущей функции, но условие возвращается другое. Обратите на использование условия 1=0, например, если атрибут RoleName имеет значение EMP. Рядовые сотрудники ничего изменять не могут. Руководители могут изменять записи своих подчиненных (но не собственную запись). Ответственные за кадры могут изменять записи всех сотрудников отдела, которым они занимаются:

```
47  function update_function(p_schema in varchar2,
48                           p_object in varchar2) return varchar2
49  is
50  begin
51       if (g_upd_pred is NULL)
52       then
53          if (sys_context(g_app_ctx, 'RoleName') = 'EMP')
54          then
55              g_upd_pred := '1=0';
56
57          elsif (sys_context(g_app_ctx, 'RoleName') = 'MGR')
58          then
59              g_upd_pred :=
60                 ' empno in (select empno
61                                from emp_base_table
62                               where mgr =
63                               sys_context(''''||g_app_ctx||
64                                           ''',"EmpNo''))';
65
66          elsif (sys_context(g_app_ctx, 'RoleName') = 'HR_REP')
67          then
68             g_upd_pred := 'deptno in
69                       (select deptno
70                          from hr_reps
71                         where username =
72                    sys_context(''''||g_app_ctx||''','',''UserName''))';
73
74          else
75              raise_application_error(-20005, 'Роль не установлена')
```

```
76          end if;
77       end if;
78
79       return g_upd_pred;
80  end;
```

Наконец, рассмотрим функцию для операторов **INSERT** и **DELETE**. В этом случае условие 1=0 возвращается для пользователей с ролями **EMP** и **MGR**: никто из них не может создавать и удалять записи — это прерогатива ответственных за кадры (пользователей с ролью **HR_REPS**):

```
81
82 function insert_delete_function(p_schema in varchar2,
83                      p_object in varchar2) return varchar2
84  is
85  begin
86      if (g_ins_del_pred is NULL)
87       then
88        if (sys_context(g_app_ctx, 'RoleName' ) in ( 'EMP', 'HGR'))
89           then
90              g_ins_del_pred := '1=0';
91         elsif (sys_context(g_app_ctx, 'RoleName') = 'HR_REP')
92           then
93             g_ins_del_pred := 'deptno in
94                    (select deptno
95                       from hr_reps
96                      where username =
97       sys_context(''''||g_app_ctx      ||'''',''UserName''))'      ;
98           else
99              raise_application_error(-20005, 'Роль не установлена');
100          end if;
101      end if;
102      return g_ins_del_pred;
103 end;
104
105 end;
106 /
```

```
Package body created.
```

До появления средств тщательного контроля доступа обеспечить применение этих трех условий при работе с одной таблицей можно было только за счет использования многочисленных представлений — по одному для операторов **SELECT**, **UPDATE** и **INSERT/DELETE** для каждой роли. Средства тщательного контроля доступа позволяют создать всего одно представление с динамически формируемыми условиями.

Последний шаг в решении задачи — связывание условий с каждой операцией ЯМД и представлением EMP. Это делается следующим образом:

```
tkyte@TKYTE816> begin
  2       dhms_rls.add_policy
  3       (object_name      => 'EMP',
  4        policy_name      => 'HR_APP_SELECT_POLICY',
  5        policy_function => 'HR_PREDICATE_PKG.SELECT_FUNCTION',
```

```
  6       statement_types => 'select');
  7  end;
  8  /
```

PL/SQL procedure successfully completed.

```
tkyte@TKYTE816>  begin
  2     dhms_rls.add_policy
  3      (object_name       => 'EMP',
  4       policy_name      => 'HR_APP_UPDATE_POLICY',
  5       policy_function  => 'HR_PREDICATE_PKG.UPDATE_FONCTION',
  6       statement_types   => 'update',
  7       update_check      => TRUE);
  8  end;
  9  /
```

PL/SQL procedure successfully completed.

```
tkyte@TKYTE816>  begin
  2     dbms_rls.add_policy
  3      (object_name      => 'EMP',
  4       policy_name     -> 'HR_APP_INSERT_DELETE_POLICY',
  5       policy_function=> 'HR_PREDICATE_PKG.INSERT_DELETE_FDNCTION',
  6       statement_types => 'insert, delete',
  7       update_check     => TRUE);
  8  end;
  9  /
```

PL/SQL procedure successfully completed.

Итак, с каждой из операций ЯМД мы связали функцию, возвращающую условие. Когда пользователь обращается с запросом к представлению **EMP,** используется условие, генерируемое функцией **HR_PREDICATE_PKG.SELECT_FUNCTION**. При изменении данных будет вызвана функция U**PDATE_FUNCTION** этого пакета и т.д.

Теперь протестируем приложение. Мы создадим пакет HR_APP. Этот пакет будет выступать в роли приложения. В нем есть подпрограммы для:

- выборки данных (процедура listEmps);
- изменения данных (процедура **updateSal);**
- удаления данных (процедура **deleteAll**);
- вставки новых данных (процедура **insertNew).**

Мы будем регистрироваться от имени различных пользователей, с разными ролями и контролировать работу приложения. В результате будет продемонстрировано, как работают средства тщательного контроля доступа.

Вот спецификация приложения:

```
tkyte@TKYTE816>create or replace package hr_app
  2  as
  3     procedure listEmps;
  4
  5     procedure updateSal;
  6
```

```
  7      procedure deleteAll;
  8
  9      procedure insertNew(p_deptno in number);
 10  end;
 11  /
Package created.
```

Теперь перейдем к телу пакета. Этот пример несколько надуманный, поскольку процедура, выполняющая **UPDATE**, пытается изменить все возможные строки, задав всем одно и то же значение. Это сделано для того, чтобы можно было точно увидеть, сколько и какие строки затрагиваются. Другие процедуры, по сути, похожи — они сообщают, что сделали и сколько строк обработано:

```
tkyte@TKYTE816> create or replace package body hr_app
  2  as
  3
  4  procedure listEmps
  5  as
  6      l_cnt number default 0;
  7  begin
  8      dbms_output.put_line
  9     (rpad('ename',10) || rpad('sal', 6 )  || ' ' ||
 10   rpad('dname',10)   ||   rpad('mgr',5)  || ' ' ||
 11       rpad('dno',3));
 12     for x in (select ename, sal, dname, mgr, emp.deptno
 13                  from emp, dept
 14                 where emp.deptno = dept.deptno)
 15     loop
 16         dbms_output.put_line(rpad(nvl(x.ename,'(null)'),10)  ||
 17                              to_char(x.sal,'9,999')  II  ' ' ||
 18                              rpad(x.dname,10)  ||
 19                             to_char(x.mgr,'9999')  ||  ' ' ||
 20                              to_char(x.deptno,'99'));
 21         l_cnt  := l_cnt + 1;
 22     end loop;
 23    dbms_output.put_line(l_cnt || ' строк(и) выбрано');
 24  end;
 25
 26
 27  procedure updateSal
 28  is
 29  begin
 30      update emp set sal = 9999;
 31     dbms_output.put_line(sql%rowcount || ' строк(и) изменено');
 32  end;
 33
 34  procedure deleteAll
 35  is
 36  begin
 37     delete from emp where empno <> sys_context('Hr_app_Ctx','EMPNO');
 38    dbms_output.put_line(sql%rowcount || ' строк(и) удалено');
```

```
39 end;
40
41  procedure insertNew(p_deptno in number)
42 as
43  begin
44    insert into emp (empno, deptno, sal) values (123, p_deptno, 1111);
45 end;
46
47  end hr_app;
48  /
Package body created.

tkyte@TKYTE816>  grant execute on hr_app to public
  2  /

Grant succeeded.
```

Вот и все наше "приложение". Процедура **listEmps** выдает все записи, доступные в представлении EMP. Процедура **updateSal** изменяет все записи, которые разрешено изменять. Процедура deleteAll удаляет все записи, которые разрешено удалять, за исключением записи текущего пользователя. Процедура **insertNew** пытается создать новую запись о сотруднике указанного отдела. Это приложение просто проверяет выполнение всех возможных операций ЯМД с представлением **EMP** (как я уже писал, приложение это весьма надуманное).

Теперь, регистрируясь от имени различных пользователей, проверим работу приложения. Сначала регистрируемся и просматриваем значения атрибутов в контексте приложения:

```
tkyte@TKYTE816>connect adams/adams

adams@TKYTE816>column namespace format a10
adams@TKYTE816>column attribute format a10
adams@TKYTE816>column value    format a10
adams@TKYTE816>select * from session_context;

NAMESPACE ATTRIBUTE VALUE


HR_APP_CTX ROLENAME  EMP
HR_APP_CTX USERNAME  ADAMS
HR_APP_CTX EMPNO     7876

adams@TKYTE816>set serveroutput on
```

Итак, поскольку мы зарегистрировались как обычный сотрудник, процедура **listEmps** должна выдать только одну запись — для этого сотрудника:

```
adams@TKYTE816>exec  tkyte.hr_app.listEmps
ename      sal    dname     mgr   dno
ADAMS       1,100 RESEARCH   7788  20
1 строк(и) выбрано

PL/SQL procedure successfully completed.
```

Поскольку мы выступаем в качестве обычного сотрудника, права изменять и удалять записи у нас быть не должно. Проверим:

```
adams@TKYTE816> exec tkyte.hr_app.updateSal
0 строк(и) изменено
PL/SQL procedure successfully completed.
adams@TKYTE816> exec tkyte.hr_app.deleteAll
0 строк(и) удалено
PL/SQL procedure successfully completed.
```

Наконец, проверим возможность выполнения оператора **INSERT**. В данном случае сервер выдаст сообщение об ошибке. В нашем примере для операторов **UPDATE** и **DELETE** ничего подобного не случилось. Попытки выполнить **UPDATE** или **DELETE** завершились успешно, потому что пользователю просто не дали данных, которые можно было бы изменить или удалить. При попытке вставки, однако, строка создается, нарушает правила зашиты, и удаляется. В этом случае сервер выдает сообщение об ошибке:

```
adams@TKYTE816> exec tkyte.hr_app.insertNew(20);
BEGIN tkyte.hr_app.insertNew(20); END;
*
ERROR at line 1:
ORA-28115: policy with check option violation
ORA-06512: at "TKYTE.HR_APP", line 36
ORA-06512: at line 1
```

Итак, мы убедились, что пользователь может просматривать только собственную запись. Попытки изменять данные любым способом, удалять и вставлять строки завершаются неудачно. Именно это и предполагалось, и обеспечивается автоматически. Приложение, пакет **HR_APP**, не содержит кода, обеспечивающего реализацию этих правил. Все они автоматически реализуются сервером, с момента начала и до завершения сеанса, независимо от того, какая среда или инструментальное средство использовано для подключения.

Теперь зарегистрируемся в качестве руководителя и посмотрим, что получится. Для начала снова получим значения атрибутов контекста, а затем — список сотрудников, записи которых доступны:

```
adams@TKYTE816>@connect jones/jones
jones@TKYTE816>set serveroutput on
jones@TKYTE816>select * from session_context;
NAMESPACE  ATTRIBUTE  VALUE

HR_APP_CTX ROLENAME   MGR
HR_APP_CTX USERNAME   JONES
HR_APP_CTX EMPNO      7566
jones@TKYTE816> exec tkyte.hr_app.listEmps
ename      sal     dname      mgr   dno
SMITH        800 RESEARCH    7902   20
JONES      2,975 RESEARCH    7839   20
SCOTT      9,999 RESEARCH    7566   20
```

```
ADAMS 1,100  RESEARCH     7788   20
FORD      3,000 RESEARCH   7566  20
5 строк(и) выбрано

PL/SQL procedure successfully completed.
```

Как видите, на этот раз мы получили из представления **EMP** несколько записей. Получены записи для всех сотрудников отдела 20 (пользователь **JONES** является его руководителем, в соответствии с информацией в представлении **EMP**). Теперь выполним процедуру изменения (**updateSal**) и проверим, какие изменения сделаны:

```
jones@TKYTE816>  exec tkyte.hr_app.updateSal
2 rows updated

PL/SQL procedure successfully completed.

jones@TKYTE816>  exec tkyte.hr_app.listEmps
ename        sal      dname        mgr     dno
SMITH          800 RESEARCH      7902    20
JONES        2,975 RESEARCH      7839    20
SCOTT        9,999 RESEARCH      7566    20
ADAMS        1,100 RESEARCH      7788    20
FORD         9,999 RESEARCH      7566    20
5 строк(и) выбрано
```

Предполагалось, что изменять можно только записи непосредственных подчиненных. Изменение затронуло только две записи для непосредственных подчиненных пользователя **JONES**. Теперь попытаемся выполнить удаление и вставку. Поскольку пользователь получил роль **MGR**, а не **HR_REP**, мы не сможем удалять записи, а при выполнении оператора **INSERT** будет получено сообщение об ошибке:

```
jones@TKYTE816> exec  tkyte.hr_app.deleteAll
0 строк(и) удалено

PL/SQL procedure successfully completed.

jones@TKYTE816> exec tkyte.hr_app.insertNew(20)
BEGIN tkyte.hr_app.insertNew(20); END;
*
ERROR at line 1:
ORA-28115: policy with check option violation
ORA-06512: at "TKYTE.HR_APP", line 44
ORA-06512: at line 1
```

Итак, руководитель может следующее.

- Просматривать не только свои собственные данные. Руководитель получает информацию обо всех своих подчиненных.
- Изменять данные. В частности, можно изменять только записи для непосредственных подчиненных, что и требовалось.
- Как и требовалось, ничего удалять и вставлять руководитель не может.

Теперь зарегистрируемся в качестве ответственного за кадры (пользователь с ролью **HR_REP**) и проверим, что позволяет сделать приложение от имени этой роли. Снова

начнем с выдачи информации о контексте приложения и списка доступных для чтения строк. На этот раз выдается вся таблица **EMP**, поскольку пользователь **KING** имеет доступ к информации по всем трем отделам:

```
jones@TKYTE816> connect king/king

king@TKYTE816> select * from session_context;

NAMESPACE  ATTRIBUTE  VALUE

HR_APP_CTX ROLENAME   HR_REP
HR_APP_CTX USERNAME   KING
HR_APP_CTX EMPNO      7839

king@TKYTE816> exec tkyte.hr_app.listEmps
ename      sal    dname      mgr   dno
CLARK       2,450 ACCOUNTING 7839  10
KING        5,000 ACCOUNTING  10
MILLER      1,300 ACCOUNTING 7782  10
SMITH         800 RESEARCH   7902  20
JONES       2,975 RESEARCH   7839  20
SCOTT       9,999 RESEARCH   7566  20
ADAMS       1,100 RESEARCH   7788  20
FORD        9,999 RESEARCH   7566  20
ALLEN       1,600 SALES      7698  30
HARD        1,250 SALES      7698  30
MARTIN      1,250 SALES      7698  30
BLAKE       2,850 SALES      7839  30
TURNER      1,500 SALES      7698  30
JAMES         950 SALES      7698  30
14 строк(и) выбрано

PL/SQL procedure successfully completed.
```

Теперь выполним изменение и посмотрим, какие данные можно изменить. В данном случае изменены все строки:

```
king@TKYTE816> exec tkyte.hr_app.updateSal
14 строк(и) изменено

PL/SQL procedure successfully completed.

king@TKYTE816> exec tkyte.hr_app.listEmps
ename      sal    dname      mgr   dno
CLARK       9,999 ACCOUNTING 7839  10
KING        9,999 ACCOUNTING  10
MILLER      9,999 ACCOUNTING 7782  10
SMITH       9,999 RESEARCH   7902  20
JONES       9,999 RESEARCH   7839  20
SCOTT       9,999 RESEARCH   7566  20
ADAMS       9,999 RESEARCH   7788  20
FORD        9,999 RESEARCH   7566  20
ALLEN       9,999 SALES      7698  30
WARD        9,999 SALES      7698  30
MARTIN      9,999 SALES      7698  30
BLAKE       9,999 SALES      7839  30
```

```
TURNER      9,999 SALES       7698  30
JAMES       9,999 SALES       7698  30
14 строк(и) выбрано

PL/SQL procedure successfully completed.
```

Значение **9,999** в столбце **SAL** доказывает, что изменены все строки таблицы. Теперь попробуем выполнить удаление. Процедура **DeleteAll** создавалась так, чтобы она не могла удалять запись текущего зарегистрированного пользователя ни при каких условиях:

```
king@TKYTE816> exec tkyte.hr_app.deleteAll
13 строк(и) удалено

PL/SQL procedure successfully completed.
```

Впервые мы смогли удалить записи. Теперь попробуем создать новую запись:

```
king@TKYTE816> exec tkyte.hr_app.insertNew(20)

PL/SQL procedure successfully completed.

king@TKYTE816> exec tkyte.hr_app.listEmps
ename       sal     dname       mgr   dno
KING        9,999 ACCOUNTING    10
(null)      1,111 RESEARCH      20
2 строк(и) выбрано

PL/SQL procedure successfully completed.
```

Понятно, что в данном случае это получится, поскольку применяются правила защиты для роли **HR_REP.** Мы завершили тестирование всех трех ролей. Все требования выполнены, данные защищены, причем защита эта реализована прозрачно для приложений.

Проблемы

Как и в случае любого средства, при использовании возможностей тщательного контроля доступа следует учитывать ряд нюансов. В этом разделе мы попытаемся их рассмотреть.

Целостность ссылок

При взаимодействии со средствами обеспечения целостности ссылок средства тщательного контроля доступа могут давать неожиданные для разработчиков результаты. Все, мне кажется, зависит от предположений о возможном взаимодействии. Лично я не вполне уверен, что при их взаимодействии должно происходить.

Оказывается, средства обеспечения целостности ссылок обходят защиту, устанавливаемую средствами тщательного контроля доступа. С их помощью я могу читать данные таблицы, удалять и изменять их, хотя непосредственно выполнять операторы **SELECT**, **DELETE** и **INSERT** для этой таблицы нельзя. Именно это и предусматривалось разработчиками СУБД и должно быть учтено при проектировании, если предполагается использовать средства тщательного контроля доступа.

Рассмотрим следующие возможности.

- Определение значений данных, которые должны быть недоступны. Это называется *тайным каналом* (covert channel). Я не могу запросить данные непосредственно. Однако, я могу доказать существование (или отсутствие) определенных значений данных в таблице, используя внешний ключ.
- Удаление данных из таблицы при наличии требования целостности ссылок с конструкцией **ON DELETE CASCADE**.
- Изменение данных в таблице при наличии требования целостности ссылок с конструкцией **ON DELETE SET NULL**.

Мы рассмотрим все три возможности на несколько надуманном примере двух таблиц — P (главная) и C (подчиненная):

```
tkyte@TKYTE816>create table p (x int primary key);

Table created.

tkyte@TKYTE816>create table c (x int references p on delete cascade);

Table created.
```

Тайный канал

Тайным каналом в данном случае называется возможность определять наличие или отсутствие значений первичного ключа в таблице P путем вставки строки в таблицу C и анализа результатов. Таким способом можно определить, есть ли в таблице P строка с соответствующим значением первичного ключа. Начнем с функции, которая всегда возвращает условие, являющееся ложным для строки:

```
tkyte@TKYTE816>create or replace function pred_function
  2  (p_schema in varchar2, p_object in varchar2)
  3  return varchar2
  4  as
  5  begin
  6          return '1=0';
  7  end;
  8  /

Function created.
```

И используем эту функцию для ограничения доступа с помощью оператора SELECT к таблице P:

```
tkyte@TKYTE816> begin
  2       dbms_rls.add_policy
  3      (object_name => 'P',
  4    policy_name    => 'P_POLICY',
  5       policy_function => 'pred_function',
  6       statement_types => 'select');
  7  end;
  8  /

PL/SQL procedure successfully completed.
```

Теперь мы по-прежнему можем вставлять значения в таблицу P (а также изменять и удалять в ней данные), но не можем ничего выбрать из этой таблицы. Начнем с вставки строки в таблицу P:

```
tkyte@TKYTE816>insert into p values (1);
1 row created.

tkyte@TKYTE816>select * from p;
no rows selected
```

Условие не позволяет выбрать эту строку, но можно проверить ее наличие, просто вставив строку в таблицу C:

```
tkyte@TKYTE816>insert into c values (1);
1 row created.
tkyte@TKYTE816>insert into c values (2);
insert into c values (2)
*
ERROR at line 1:
ORA-02291: integrity constraint (TKYTE.SYS_C003873) violated - parent key
not found
```

Итак, мы теперь знаем, что значение 1 содержится в таблице P, а значения 2 в ней нет, потому что в таблицу C удалось вставить строку со значением 1 и не удалось — со значением 2. Средства обеспечения целостности ссылок могут читать данные, несмотря на установленные средствами тщательного контроля доступа правила защиты. Это может стать сюрпризом для приложения, например, средства, генерирующего запросы на основе информации из словаря данных. Если запросить данные из таблицы C, все необходимые строки возвращаются. При попытке же соединения таблиц P и C будет получено пустое результирующее множество.

Следует также отметить, что подобный тайный канал получения данных из подчиненной таблицы возможен и для главной таблицы. Если бы аналогичное правило защиты было установлено не для таблицы P, а для C и для нее не была бы задана конструкция **ON DELETE CASCADE** (другими словами, требовалось бы только наличие соответствующего значения первичного ключа), можно было бы определить, какие значения столбца X есть в таблице C, удаляя строки из таблицы P. Если в подчиненной таблице C есть строки с соответствующим значением, попытка удаления строки из таблицы P приведет к выдаче сообщения об ошибке, а если — нет, удаление пройдет успешно, хотя выбирать строки из таблицы C с помощью SELECT и нельзя.

Удаление строк

Это можно сделать при наличии конструкции ON DELETE CASCADE в требовании целостности. Если удалить правило для таблицы P и задать эту же функцию в качестве правила защиты для удаления из таблицы C следующим образом:

```
tkyte@TKYTE816> begin
  2   dbms_rls.drop_policy
  3  ('TKYTE',  'P', 'P_POLICY');
  4  end;
```

```
  5  /

PL/SQL procedure successfully completed.
tkyte@TKYTE816> begin
  2       dbms_rls.add_policy
  3      (object_name => 'C',
  4       policy_name => 'C_POLICY',
  5       policy_function => 'pred_function',
  6       statement_types => 'DELETE');
  7  end;
  8  /

PL/SQL procedure successfully completed.
```

то окажется, что нельзя удалить **ни одной** строки из таблицы C с помощью SQL-оператора:

```
tkyte@TKYTE816>delete from C;

0 rows deleted.
```

Установленное правило защиты не позволяет это сделать. Наличие строки в таблице C (в результате вставки в предыдущем примере) легко проверить:

```
tkyte@TKYTE816>select * from C;

         X

         1
```

Простое удаление строки в главной таблице:

```
tkyte@TKYTE816>delete from P;

1 row deleted.
```

снова позволяет обойти правило защиты, задаваемое средствами тщательного контроля доступа — соответствующая строка из таблицы C тоже автоматически удаляется:

```
tkyte@TKYTE816>select * from C;

no rows selected
```

Изменение строк

Аналогичная ситуация при удалении строк из главной таблицы возникает и при использовании в требовании целостности конструкции **ON DELETE SET NULL.** Немного изменим пример, чтобы, благодаря требованию целостности ссылок, можно было изменять строки в таблице C, которые нельзя изменять SQL-операторами. Начнем с пересоздания таблицы C, задав для внешнего ключа конструкцию **ON DELETE SET NULL**.

```
tkyte@TKYTE816>drop table c;

Table dropped.

tkyte@TKYTE816> create table c (x int references p on delete set null);

Table created.
```

```
tkyte@TKYTE816> insert into p values (1);
1 row created.
tkyte@TKYTE816> insert into c values (1);
1 row created.
```

Теперь зададим ту же функцию, что и в предыдущих примерах, в качестве правила защиты для изменения данных в таблице C, и установим флагу **UPDATE_CHECK** значение TRUE. Это не позволит изменять строки:

```
tkyte@TKYTE816> begin
  2     dbms_rls.add_policy
  3     (object_name -> 'C',
  4      policy_name => 'C_POLICY',
  5      policy_function => 'pred_function',
  6      statement_types => 'UPDATE',
  7      update_check => TRUE);
  8  end;
  9  /
PL/SQL procedure successfully completed.
tkyte@TKYTE816> update c set x = NULL;
0 rows updated.
tkyte@TKYTE816> select * from c;
         X
         1
```

Итак, с помощью SQL-операторов строки в таблице C изменять нельзя. Однако удаление строк из таблицы P показывает следующее:

```
tkyte@TKYTE816> delete from p;
1 row deleted.
tkyte@TKYTE816> select * from c;
         X
```

Итак, обходным путем можно изменить данные в таблице C. Есть и другой способ продемонстрировать это, но данные в таблицах придется восстановить:

```
tkyte@TKYTE816> delete from c;
1 row deleted.
tkyte@TKYTE816> insert into p values (1);
1 row created.
tkyte@TKYTE816> insert into c values (1);
1 row created.
```

Теперь перепишем функцию так, чтобы можно было изменять строки в таблице C, задавая им любые значения, кроме Null:

```
tkyte@TKYTE816>create or replace function pred_function
  2  (p_schema in varchar2, p_object in varchar2)
  3  return varchar2
  4  as
  5  begin
  6          return 'x is not null';
  7  end;
  8  /

Function created.

tkyte@TKYTE816>update c set x = NULL;
update c set x = NULL
       *
ERROR at line 1:
ORA-28115: policy with check option violation
```

Это изменение завершилось неудачно, поскольку условие **X IS NOT NULL** не будя выполняться после изменения. Если теперь снова удалить строку из таблицы P:

```
tkyte@TKYTE816>delete from p;

1 row deleted.

tkyte@TKYTE816>select * from c;

         X
```

строка в таблице C получит значение, которое нельзя задать с помощью SQL-оператора непосредственно.

Кэширование курсоров

Одна из важных особенностей реализации функции, задающей правила защиты, из первого примера данной главы состоит в том, что в ходе сеанса эта функция возвращает одно и то же условие. Это принципиально важно. Если еще раз рассмотреть алгоритм этой функции:

```
  5  as
  6  begin
  7     if (user = p_schema) then
  8         return '';
  9      else
 10         return 'owner = USER'
 11      end if;
 12  end;
```

оказывается, что она возвращает либо пустое условие, либо условие **owner = USER**. Но в течение сеанса она постоянно возвращает одно и то же условие. Невозможно получить условие owner = USER, а затем в том же сеансе получить пустое условие. Чтобы понять, почему это принципиально важно для корректного использования средств тщательного контроля доступа в приложении, надо разобраться, когда условие связывается

с запросом и как это происходит при использовании различных сред: PL/SQL, Pro*C, OCI, JDBC, ODBC и т.д.

Предположим, имеется следующая функция, возвращающая условие:

```
SQL> create or replace function rls_examp
  2  (p_schema in varchar2, p_object in varchar2)
  3  return varchar2
  4  as
  5  begin
  6         if (sys_context('myctx', 'x') is not null)
  7          then
  8                 return 'x > 0';
  9          else
 10                 return '1=0';
 11          end if;
 12  end;
 13  /

Function created.
```

Она реализует такой алгоритм: если в контексте установлен атрибут x, возвращается условие x > 0; если же в контексте атрибут x не установлен, возвращается условие 1=0. Если создать таблицу T, поместить в нее данные и добавить правила и контекст следующим образом:

```
SQL> create table t (x int);

Table created.

SQL> insert into t values (1234);

1 row created.

SQL> begin
  2     dbms_rls.add_policy
  3     (object_schema   => user,
  4      object_name => 'T',
  5      policy_name => 'T_POLICY',
  6      function_schema => user,
  7      policy_function => 'rls_examp',
  8      statement_types => 'select');
  9  end;
 10  /

PL/SQL procedure successfully completed.

SQL> create or replace procedure set_ctx(p_val in varchar2)
  2  as
  3  begin
  4          dbms_session.set_context('myctx', 'x', p_val);
  5  end;
  6  /

Procedure created.

SQL> create or replace context myctx using set_ctx;

Context created.
```

Предполагается, что в случае установки контекста мы должны получить одну строку. Если же контекст не установлен, мы ни одной строки получить не должны. Если проверить это в среде SQL*Plus с помощью простых SQL-операторов, именно так и окажется:

```
SQL> exec set_ctx(null);
PL/SQL procedure successfully completed.
SQL> select * from t;
no rows selected
SQL> exec set_ctx(1);
PL/SQL procedure successfully completed.
SQL> select * from t;
         X

      1234
```

Итак, казалось бы, все в порядке. Динамически формируемое условие применяется так, как предполагалось. Фактически же, если использовать язык PL/SQL (или Pro*C, или правильно написанное приложение, использующее интерфейсы OCI/JDBC/ODBC, да и многие другие среды), оказывается, что это не так. Создадим, например, небольшую PL/SQL-процедуру:

```
SQL> create or replace procedure dump_t
  2  (some_input in number default NULL)
  3  as
  4  begin
  5          dbms_output.put_line
  6      ('***      Результат выполнения оператора SELECT * FROM T');
  7
  8          for x in (select * from t) loop
  9                  dbms_output.put_line(x.x);
 10          end loop;
 11
 12          if (some_input is not null)
 13          then
 14                  dbms_output.put_line
 15          ('***          Результат выполнения другого оператора SELECT
-> * FROM T');
 16
 17                  for x in (select * from t) loop
 18                          dbms_output.put_line(x.x);
 19                  end loop;
 20          end if;
 21  end;
 22  /
Procedure created.
```

Эта процедура выполняет оператор **SELECT * FROM T** один раз, если входные данные не переданы, и два раза, если переданы какие-либо входные данные. Выполним эту

процедуру и посмотрим результаты. Выполнение процедуры начнем, установив в контексте значение Null (поэтому будет применяться условие 1—0, другими словами, не будет возвращена ни одна строка):

```
SQL> set serveroutput on
SQL> exec set_ctx(NULL)
PL/SQL procedure successfully completed.

SQL> exec dump_t
*** Результат выполнения оператора SELECT * FROM T
PL/SQL procedure successfully completed.
```

Как и ожидалось, данные не получены. Теперь установим значение в контексте так, чтобы возвращалось условие x > 0. Затем вызовем процедуру DUMP_T так, чтобы она выполняла оба запроса. В версиях Oracle 8.1.5 и 8.1.6 при этом произойдет следующее:

```
SQL> exec set_ctx(1)
PL/SQL procedure successfully completed.

SQL> exec dump_t(0)
*** Результат выполнения оператора SELECT * FROM T
*** Результат выполнения другого оператора SELECT * FROM T
1234
PL/SQL procedure successfully completed.
```

Первый запрос, первоначально выполненный при значении **Null** в контексте, по-прежнему **не возвращает данных.** Его курсор был сохранен в кэше и повторно не анализировался.

При выполнении процедуры со значением **Null** атрибута x в контексте, получаем предполагаемые результаты (потому что это было первое выполнение данной процедуры в сеансе). Устанавливаем атрибуту x непустое значение и получаем неоднозначные результаты. Первый оператор **SELECT * FROM T** в процедуре по-прежнему не возвращает ни одной строки — он, видимо, продолжает использовать условие 1=0. Второй запрос (который первый раз мы не выполняли) возвращает, как и предполагалось, правильные результаты. Он использует условие x > 0, как и было задумано.

Почему первый оператор **SELECT** в этой процедуре не использует условие, которое мы предполагали? Это связано с оптимизацией, называемой *кэшированием курсора*. Язык PL/SQL и многие другие среды выполнения не "закрывают" курсор, когда этого требует разработчик. Представленный пример можно легко воспроизвести в Pro*C, если оставить опции прекомпилятора **release_cursor** стандартное значение **NO**. Если тот же код обработать с опцией **release_cursor=YES,** программа Pro*C будет работать аналогично запросам в среде SQL*Plus. Условие, используемое пакетом **DBMS_RLS**, связывается с запросом на стадии анализа. Первый запрос **SELECT * FROM T** анализируется при первом выполнении хранимой процедуры, когда фактически возвращалось условие 1=0. PL/SQL-машина автоматически кэширует проанализированный курсор. При втором выполнении хранимой процедуры PL/SQL-машина повторно использует проанализированный курсор первого запроса **SELECT * FROM T.** Этот проанализированный курсор

включает условие 1=0. Функция, возвращающая условие, на этот раз вообще не вызывалась. Поскольку мы передали процедуре входные данные, PL/SQL-машина выполнила и второй запрос. Для этого запроса, однако, еще нет открытого и проанализированного курсора, так что при выполнении он был проанализирован, с учетом непустого атрибута в контексте. Со вторым запросом **SELECT * FROM T** было связано условие x>0. Это и вызвало двусмысленность. Поскольку мы не можем управлять кэшированием курсора, возвращения в одном сеансе различных условий функцией, реализующей правила защиты, надо избегать любой ценой. В противном случае возможны плохо воспроизводимые и сложные для отладки ошибки в приложении. Ранее, в примере приложения для работы с информацией о сотрудниках, было продемонстрировано, как реализовать функцию, которая возвращает в ходе сеанса не более одного условия. Это гарантирует следующее.

- Согласованность результатов запросов с точки зрения средств тщательного контроля доступа.
- Неизменность условий защиты по ходу сеанса. Если они изменяются, возможны странные и непредсказуемые результаты.
- Зависимость правил защиты от пользователя, выполняющего операторы, но не от среды, в которой он работает.

В версиях Oracle 8.1.7 и выше результат будет следующим:

```
tkyte@dev817>exec dump_t(0)
*** Результат выполнения оператора SELECT * FROM T
1234
*** Результат выполнения другого оператора SELECT * FROM T
1234

PL/SQL procedure successfully completed.
```

Во избежание описанных проблем сервер Oracle версий 8.1.7 и выше повторно анализирует запрос при изменении контекста сеанса, если с запросом связаны правила защиты. Подчеркиваю: **при изменении контекста сеанса.** Если для создания условия не используется контекст сеанса, снова возникает проблема, связанная с кэшированием курсора. Рассмотрим систему, в которой условия хранятся как данные в таблице базы данных. Такая система реализует правила защиты на основе таблицы. Если при этом содержимое таблицы правил изменится, вызывая изменение условия, мы столкнемся» версии 8.1.7 с теми же проблемами, что и в 8.1.6, и более ранних версиях. Если изменить предыдущий пример так, чтобы использовалась таблица базы данных:

```
tkyte@TKYTE816>create table policy_rules_table
  2  (predicate_piece varchar2(255)
  3  );

Table created.

tkyte@TKYTE816>insert into policy_rules_table values ('x > 0');

1 row created.
```

и изменить функцию, реализующую правила защиты так, чтобы она использовала таблицу:

```
tkyte@TKYTE816>create or replace function rls_examp
  2  (p_schema in varchar2, p_object in varchar2)
  3  return varchar2
  4  as
  5     l_predicate_piecevarchar2(255);
  6  begin
  7     select predicate_piece into l_predicate_piece
  8       from policy_rules_table;
  9
 10     return l_predicate_piece;
 11  end;
 12  /

Functioncreated.
```

то можно ожидать следующих результатов выполнения процедуры **DUMP_T (при** изменении условия **после выполнения DUMP_T** без параметров, **но до** выполнения **ее с** параметром):

```
tkyte@DEV817>  exec dump_t
*** Результат выполнения оператора SELECT * FROM T
1234

PL/SL procedure successfully completed.

tkyte@DEV817> update policy_rules_table set predicate_piece = '1=0';

1 row updated.

tkyte@DEV817>exec dump_t(0)
*** Результат выполнения оператора SELECT * FROM T
1234
*** Результат выполнения другого оператора SELECT * FROM T

PL/SQL procedure successfully completed.
```

Обратите внимание, что при первом выполнении использовалось условие x>0, и запрос возвращал одну строку из таблицы T. После выполнения этой процедуры мы изменили условие (это изменение можно было сделать из другого сеанса — его, например, мог сделать администратор). При втором выполнении процедуры **DUMP_T** — с параметром, требующим выполнить после первого запроса второй, оказывается, что в первом запросе по-прежнему используется старое условие, x>0, тогда как во втором запросе — второе условие, 1=0, только что помещенное в таблицу **POLICY_RULES.** Необходимо учитывать последствия кэширования курсора, даже в версиях 8.1.7 и выше, если только в правилах защиты наряду с таблицей не используется контекст приложения.

Хочу подчеркнуть, что изменять значение SYS_CONTEXT по ходу работы приложения **вполне допустимо.** Эти изменения будут учтены и использованы при следующем выполнении запроса. Поскольку значения атрибутов контекста передаются как связываемые переменные, они вычисляются на этапе выполнения запроса, а не на этапе анализа, так что константы на этапе анализа не используются. Только текст условия не должен меняться по ходу работы приложения. Вот небольшой пример, демонстрирующий это. Завершим сеанс и снова зарегистрируемся (чтобы очистить кэш курсоров в сеансе), а затем изменим реализацию функции **RLS_EXAMP.** Затем выполним те же действия, что и раньше:

```
tkyte@TKYTE816>connect tkyte/tkyte

tkyte@TKYTE816>create or replace function rls_examp
  2  (p_schema in varchar2, p_object in varchar2)
  3  return varchar2
  4  as
  5  begin
  6          return  'x > sys_context(''myctx'',''x'')';
  7  end;
  8  /

Function created.

tkyte@TKYTE816>set serveroutput on

tkyte@TKYTE816>exec set_ctx(NULL)

PL/SQL procedure successfully completed.

tkyte@TKYTE816>exec dump_t
*** Результат выполнения оператора SELECT * FROM T

PL/SQL procedure successfully completed.

tkyte@TKYTE816>exec set_ctx(1)

PL/SQL procedure successfully completed.

tkyte@TKYTE816>exec dump_t(0)
*** Результат выполнения оператора SELECT * FROM T
1234
*** Результат выполнения другого оператора SELECT * FROM T
1234

PL/SQL procedure successfully completed.
```

На этот раз оба запроса вернули одинаковые результаты. Это связано с тем, что они используют одинаковую конструкцию **WHERE** и в условии динамически обращаются к значению атрибута в контексте приложения.

Следует упомянуть, что бывают случаи, когда изменение условия по ходу сеанса может оказаться необходимым. Для этого приходится специальным образом программировать приложения, обращающиеся к объектам, правила защиты которых допускают изменение условий по ходу сеанса. Например, в PL/SQL придется использовать в приложении исключительно динамический SQL, чтобы избежать кэширования курсоров. При использовании этого способа поддержки динамических условий следует помнить, что результаты будут зависеть от того, как написано клиентское приложение, так что подобным образом универсальные правила защиты реализовать не удастся. Мы не будем рассматривать этот способ использования средств пакета **DBMS_RLS**, а сосредоточимся на его исходном назначении — защите данных от несанкционированного доступа.

Экспортирование/Импортирование

Эту проблему мы уже упоминали. Необходимо быть внимательным при использовании утилиты **EXP** для экспортирования данных и утилиты **IMP** для их импортирования. Поскольку при этом возникают разные проблемы, они будут рассмотрены по от-

дельности. Чтобы продемонстрировать проблемы, придется несколько расширить предыдущий пример, изменив правила защиты **T_POLICY**. На этот раз правила будут применяться не только для операторов **SELECT**, но и для операторов **INSERT**:

```
tkyte@TKYTE816> begin
  2   dbms_rls.drop_policy('TKYTE',   'T', 'T_POLICY');
  3  end;
  4  /
PL/SQL procedure successfully completed.

tkyte@TKYTE816> begin
  2     dbms_rls.add_policy
  3     (object_name => 'T',
  4      policy_name => 'T_POLICY',
  5      policy_function => 'rls_examp',
  6      statement_types => 'select, insert',
  7       update_check    => TRUE);
  8  end;
  9  /
PL/SQL procedure successfully completed.
```

После этого мы получим следующий эффект:

```
tkyte@TKYTE816>  delete from t;

1 row deleted.

tkyte@TKYTE816> commit;

Commit complete.

tkyte@TKYTE816> exec set_ctx(null);

PL/SQL procedure successfully completed.

tkyte@TKYTE816> insert into t values (1);
insert into t values (1)
            *
ERROR at line 1:
ORA-28115: policy with check option violation

tkyte@TKYTE816> exec set_ctx(0) ;

PL/SQL procedure successfully completed.

tkyte@TKYTE816> insert into t values (1);

1 row created.
```

Итак, теперь контекст необходимо устанавливать не только для чтения, но и для вставки данных.

Проблемы экспорта

Стандартно утилита **EXP** работает в режиме "обычного" (conventional path) экспорта. Для чтения данных она использует SQL-операторы. Используя утилиту **EXP** для получения данных таблицы T из базы данных, получим следующий результат (учтите, что в таблице T имеется одна строка — результат выполнения оператора **INSERT**):

```
C:\fgac\exp userid=tkyte/tkyte tables=t

Export: Release 8.1.6.0.0 - Production on Mon Apr 16 16:29:25 2001
(c) Copyright 1999 Oracle Corporation.  All rights reserved.


Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option
JServer Release 8.1.6.0.0 — Production
Export done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR
character set

About to export specified tables via Conventional Path ___
EXP-00079: Data in table "T" is protected. Conventional path may only be
exporting partial table.
. . exporting table                                T        0 rows exported
Export terminated successfully with warnings.
```

Обратите внимание, как утилита EXP "великодушно" сообщила, что таблица может быть экспортирована только частично, поскольку используется обычный способ экспортирования. Для решения этой проблемы при экспортировании надо подключаться как пользователь SYS (или от имени любого другого пользователя с ролью SYSDBA). Для пользователя SYS средства тщательного контроля доступа не действуют:

```
C:\fgac\exp userid=sys/manager tables=tkyte.t

Export: Release 8.1.6.0.0 - Production on Mon Apr 16 16:35:21 2001
(c) Copyright 1999 Oracle Corporation. All rights reserved.


Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option
JServer Release 8.1.6.0.0 - Production
Export done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR
character set

About to export specified tables via Conventional Path ...
Current user changed to TKYTE
. . exporting table                                T        1 rows exported
Export terminated successfully without warnings.
```

Можно также использовать процедуру **DBMS_RLS.ENABLE_POLICY** для временного отключения правил зашиты и повторного их включения после экспортирования. Хотя так делать нежелательно, поскольку на время экспорта таблицы остаются незащищенными.

В некоторых версиях Oracle 8.1.5 при непосредственном экспорте по ошибке обходятся средства тщательного контроля доступа. Другими словами, если указать опции ***direct=true,*** *будут экспортированы все данные. На этот способ полагаться не стоит поскольку в следующих версиях эта ошибка была исправлена. В новых версиях ее получите:*

```
About to export specified tables via Direct Path ...
EXP-00080: Data in table "T" is protected. Using conventional mode.
EXP-00079: Data in table "T" is protected. Conventional path may only...
```

Утилита EXP будет автоматически экспортировать защищенные таблицы в обычном режиме.

Проблемы импорта

Эти проблемы возникают только в том случае, когда установлены правила защиты для операторов вставки и опция UPDATE_CHECK имеет значение True. В этом случае утилита IMP не будет вставлять строки, не удовлетворяющие условию, возвращаемому соответствующей функцией. Именно так и произойдет в рассмотренном ранее примере. Если не установить контекст, ни одна строка вставлена не будет (по умолчанию значение в контексте — Null). Поэтому, если попытаться импортировать экспортированные данные:

```
C:\fgac>imp userid=tkyte/tkyte full=y ignore=y

Import: Release 8.1.6.0.0 - Production on Mon Apr 16 16:37:33 2001

(c) Copyright 1999 Oracle Corporation.  All rights reserved.


Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 — Production
With the Partitioning option

JServer Release 8.1.6.0.0 — Production

Export file created by EXPORT:V08.01.06 via conventional path

Warning: the objects were exported by SYS, not by you
import done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR
character set
. importing SYS's objects into TKYTE
. . importing table                            "T"
IMP-00058: ORACLE error 28115 encountered
ORA-28115: policy with check option violation
IMP-00017: following statement failed with ORACLE error 28101:
 "BEGIN   DBMS_RLS.ADD_POLICY('TKYTE',
'T', 'T_POLICY','TKYTE','RLS_EXAMP', 'SE"
 "LECT,INSERT',TRUE,TRUE); END;"
IMP-00003: ORACLE error 28101 encountered
ORA-28101: policy already exists
ORA-06512: at "SYS.DBMS_RLS", line 0
ORA-06512: at line 1
import terminated successfully with warnings.
```

строки вставлены не будут. Проблему можно решить, подключившись от имени пользователя SYS или пользователя с ролью SYSDBA:

```
C:\fgac>imp userid=sys/manager full=y ignore=y

import: Release 8.1.6.0.0 - Production on Mon Apr 16 16:40:56 2001

(c) Copyright 1999 Oracle Corporation.  All rights reserved.


Connected to: Oracle8i Enterprise Edition Release 8.1.6.0.0 - Production
With the Partitioning option
JServer Release 8.1.6.0.0 - Production
```

```
Export file created by EXPORT:V08.01.06 via conventional path
import done in WE8ISO8859P1 character set and WE8ISO8859P1 NCHAR
character set
. importing SYS's objects into SYS
. importing TKYTE's objects into TKYTE
. . importing table                            "T"          1 rows imported
```

Можно также с помощью процедуры DBMS_RLS.ENABLE_POLICY временно отключить правила и включить их после импортирования. Как и в случае экспорта, это не желательно, поскольку в процессе импортирования таблица не защищена.

Отладка

При написании функций, возвращающих условия, я часто использую пакет средств отладки, debug. Этот пакет, созданный сотрудником корпорации Oracle Кристофером Беком (Christopher Beck), позволяет включить в код операторы отладочной печати. Он позволяет свободно вставлять в код операторы вида:

```
create function foo ...
as

begin
     debug.f('Входим в процедуру foo');
     if (some_condition) then
         l_predicate := 'x=1';
     end if;

     debug.f('Возвращаем условие ''%s''', l_predicate);
     return l_predicate;
end;
```

Итак, процедура **debug.f** работает аналогично С-функции **printf** и реализована с помощью средств пакета **UTL_FILE**. Она создает управляемые программистом файлы трассировки на сервере базы данных. Файлы трассировки содержат результаты отладочной печати, благодаря которым можно понять, как выполняется код. Поскольку ядро сервера вызывает код реализующих правила защиты функций в фоновом режиме, отладка их затруднена. Традиционные средства вроде пакета **DBMS_OUTPUT** и отладчика PL/SQL тут не особенно помогут. При наличии файлов трассировки можно сэкономить очень много времени. Среди сценариев, которые можно загрузить с сайта издательства Wrox, находится и пакет **debug** с комментариями по его настройке и использованию.

Этот пакет особенно полезен при поиске проблем в функциях, реализующих правила защиты, и я настоятельно рекомендую использовать его или другой подобный пакет. При отсутствии подобных средств трассировки практически невозможно разобраться, что работает неправильно.

Ошибки, которые могут произойти

По ходу реализации рассмотренного ранее приложения я столкнулся с многочисленными ошибками, и мне пришлось заниматься его отладкой. Поскольку средства тщательного контроля доступа работают на сервере, поиск причин ошибки и отладка при-

ложения очень затруднена. Изучив следующие разделы, вы сможете находить и успешно устранять причины ошибок*.

ORA-28110: пакет или функция <имя функции> методики имеет ошибку

Эта ошибка свидетельствует о том, что в пакете или функции, реализующей правила защиты, имеется ошибка, и перекомпиляция ее невозможна. Если выполнить в среде SQL*Plus команду **SHOW ERRORS FUNCTION <ИМЯ ФУНКЦИИ>** или **SHOW ERRORS PACKAGE BODY <ИМЯ ПАКЕТА>,** можно получить соответствующие сообщения об ошибках.

Эта ошибка может быть связана с тем, что:

- один из объектов, на который ссылается функция, удален или стал недействительным;
- в компилируемом коде есть синтаксическая ошибка или его по какой-то причине нельзя скомпилировать.

Наиболее типичная причина этой ошибки состоит в том, что условие, которое возвращает функция, реализующая правила защиты для таблицы, содержит ошибку. Рассмотрим функцию, использованную в предыдущих примерах:

```
tkyte@TKYTE816> create or replace function rls_examp
  2  (p_schema in varchar2, p_object in varchar2)
  3  return varchar2
  4  as
  5  begin
  6          this is an error
  7          return 'x > sys_context(''myctx'',''x'')';
  8  end;
  9  /
```

Warning: Function created with compilation errors.

Предположим, при компиляции мы не обратили внимания, что функция не откомпилирована, как положено. Мы предполагаем, что функция скомпилирована успешно, и что ее можно выполнять. Теперь, попытавшись выполнить запрос к таблице T, мы получим:

```
tkyte@TKYTE816>  exec set_ctx(0);
PL/SQL procedure successfully completed.
tkyte@TKYTE816>  select * from t;
select * from t
              *
ERROR at line 1:
ORA-28110: policy function or package TKYTE.RLS_EXAMP has error
```

' *Тексты сообщений об ошибках в названиях подразделов здесь представлены так, как их выдает сервер Oracle 8.1.6 при установке русского в качестве языка сообщений. В примерах кода оставлены сообщения об ошибках на английском. Обратите внимание на расхождениях терминологии: "методика" вместо "правила".* - Прим. научн. ред.

Итак, это сообщение свидетельствует о наличии ошибки, а именно, об ошибке в функции **TKYTE.RLS_EXAMP** (ее не удалось успешно скомпилировать). Для того чтобы находить подобные проблемы прежде, чем они возникнут, пригодится следующий запрос:

```
tkyte@TKYTE816> column pf_owner format a10
tkyte@TKYTE816> column package format a10
tkyte@TKYTE816> column function format a10
tkyte@TKYTE816> select pf_owner, package, function
  2    from user_policies a
  3   where exists (select null
  4                   from all_objects
  5                  where owner = pf_owner
  6                    andobject_type in ('FUNCTION', 'PACKAGE',
  7                                       'PACKAGE BODY')
  8                    and status = 'INVALID'
  9                   and object_name in (a.package, a.function)
 10                )
 11  /

PF_OWNER   PACKAGE    FUNCTION

TKYTE                 RLS_EXAMP
```

Этот запрос выдает список всех **недопустимых** функций, реализующих правила защиты. Пока что он подтверждает то, что мы и так знаем, — что функция **TKYTE.RLS_EXAMP** скомпилирована с ошибками. Решить эту проблему несложно. Выполняем:

```
tkyte@TKYTE816> show errors function rls_examp
Errors for FUNCTION RLS_EXAMP:

LINE/COL ERROR

6/10    PLS-00103: Encountered the symbol "AN" when expecting one of the
          following:
          := . ( @ % ;
```

Исправляем ошибку в строке 6 с текстом this is an error, и сообщение **ORA-28110** выдаваться не будет.

ORA-28112: сбой при выполнении функции методики

Сообщение об ошибке ORA-28112: failed to execute policy function выдается при выполнении оператора SELECT или оператора ЯМД для таблицы, относительно которой заданы правила защиты, если в реализующей эти правила функции (а не в возвращаемом ею условии) произошла ошибка. Это означает, что при выполнении функции возбуждена исключительная ситуация, не обработанная в самой функции и полученная ядром сервера.

При возникновении ошибки ORA-28112 в каталоге, заданном параметром инициализации USER_DUMP_DEST, будет генерироваться файл трассировки. В этом файле не будет сообщения об ошибке ORA-28112, но будет фраза Policy function execution error.

Зададим для функции следующий алгоритм (продолжаем предыдущий пример):

```
tkyte@TKYTE816> create or replace function rls_examp
  2  (p_schema in varchar2, p_object in varchar2)
  3  return varchar2
  4  as
  5          l_uid number;
  6  begin
  7          select user_id
  8            into l_uid
  9            from all_users
 10           where username -  'ИМЯ_НЕСУЩЕСТВУКЩЕГО_ПОЛЬРЗОВАТЕЛЯ';
 11
 12     return 'x > sys_context(''myctx'',''x'')';
 13  end;
 14  /

Function created.
```

Эта функция специально написана так, чтобы возбуждалась и не обрабатывалась исключительная ситуация **NO_DATA_FOUND**. Любопытно посмотреть, что произойдет, если исключительная ситуация распространится на уровень ядра сервера. Инициируем вызов этой функции:

```
tkyte@TKYTE816> exec set_ctx(0);

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select * from t;
select * from t
              *
ERROR at line 1:
ORA-28112: failed to execute policy function
```

Это означает, что функция, реализующая правила защиты, существует и синтаксически корректна, но при ее выполнении возбуждается исключительная ситуация. При возникновении этой ошибки создается файл трассировки. Если посмотреть содержимое каталога, задаваемого параметром инициализации **USER_DUMP_DEST** и найти этот файл трассировки, в самом начале можно обнаружить следующее:

```
*** SESSION ID:(8.405) 2001-04-16 17:03:00.193
*** 2001-04-16 17:03:00.193

Policy function execution error:
Logon user     : TKYTE
Table or View  : TKYTE.T
Policy name    : T_POLICY
Policy function: TKYTE.RLS_EXAMP
ORA-01403: no data found
ORA-06512: at "TKYTE.RLS_EXAMP", line 7 .
ORA-06512: at line 1
```

Эта информация позволяет определить, в каком месте функции произошла ошибка. Явно сказано про строку 7, содержащую оператор **SELECT ... INTO**, а также указано, что в этой строке возбуждена исключительная ситуация **NO_DATA_FOUND**.

ORA-28113: ошибка в предикате методики

Сообщение об ошибке **ORA-28113: policy predicate has error** выдается при выполнении оператора **SELECT** или оператора ЯМД для таблицы, относительно которой заданы правила защиты, если соответствующая функция возвращает синтаксически или семантически неверное условие. Это условие при добавлении к исходному запросу дает синтаксически неправильный SQL-оператор.

При возникновении ошибки **ORA-28113** в каталоге, заданном параметром инициализации **USER_DUMP_DEST**, генерируется файл трассировки. В нем будет сообщение об ошибке **ORA-28113** и информация о текущем сеансе и ошибочном условии.

Пусть, например, функция реализована так, как показано ниже. Она возвращает условие, сравнивающее значение в столбце X с несуществующим столбцом таблицы:

```
tkyte@TKYTE816> create or replace function rls_examp
  2  (p_schema in varchar2, p_object in varchar2)
  3  return varchar2
  4  as
  5  begin
  6      return 'x = несуществующий_столбец';
  7  end;
  8  /

Function created.
```

Так что запрос вида:

```
select * from t
```

будет переписан как:

```
select * from (select * from t where x = несуществующий_столбец)
```

Очевидно, поскольку в таблице T такого столбца нет, этот запрос выполнить нельзя.

```
tkyte@TKYTE816> select * from t;
select * from t
              *
ERROR at line 1:
ORA-28113: policy predicate has error
```

Функция успешно вернула условие, но при добавлении этого условия к запросу произошла ошибка. В конце текста соответствующего файла трассировки на сервере мы обнаружим:

```
*** SESSION ID:(8.409) 2001-04-16 17:08:10.669
*** 2001-04-16 17:08:10.669


Error information for ORA-28113:
Logon user     : TKYTE
Table or View  : TKYTE.T
Policy name    : T_POLICY
Policy function: TKYTE.RLS_EXAMP
RLS predicate  :
```

```
x = несуществукщий_столбец
ORA-00904: invalid column name
```

Этой информации достаточно для решения проблемы (изменения условия, которое привело к выдаче сообщения об ошибке), поскольку, помимо ошибочного условия, имеется сообщение об ошибке в этом условии.

ORA-28106: вводимое значение для аргумента #2 неверно

Если имя атрибута не является допустимым идентификатором Oracle, сообщение об ошибке можно получить при вызове процедуры **DBMS_SESSION.SET_CONTEXT.** Атрибуты в контексте приложения должны именоваться с соответствии с соглашениями, принятыми в Oracle (точно так же, как имена столбцов таблиц или переменных PL/SQL). Единственное решение — изменить имя атрибута. Нельзя, например, использовать атрибут контекста с именем SELECT, придется переименовать его.

Резюме

В этой главе были подробно рассмотрены средства тщательного контроля доступа. Есть много аргументов за использование этих средств и лишь несколько — против. В общем-то, найти аргументы против использования этих средств непросто. Мы рассмотрели, как средства тщательного контроля доступа:

- **Упрощают разработку приложений.** Они отделяют управление доступом от приложения, приближая его к данным.
- **Гарантируют постоянную защиту данных.** Независимо от того, какое средство использовано для доступа к данным, правила защиты применяются неукоснительно, и обойти их нельзя.
- **Позволяют изменять правила защиты без изменения клиентских приложений.**
- **Упрощают управление объектами базы данных.** Их использование позволяет уменьшить количество объектов базы данных, необходимых для поддержки приложения.

Эти средства обеспечивают отличную производительность. Производительность фактически зависит от производительности алгоритмов и SQL-операторов, реализующих правила защиты. Если возвращаемое условие не позволяет оптимизатору выработать эффективный план выполнения, это никак не связано со средствами тщательного контроля доступа, а исключительно с настройкой SQL-оператора. Использование контекста приложения позволяет воспользоваться всеми преимуществами разделяемых SQL-операторов и уменьшить количество создаваемых для базы данных объектов. Средства тщательного контроля доступа не снижают производительность в большей степени, чем любой другой способ выполнения тех же проверок.

Было также показано, что эти средства могут создавать определенные проблемы при отладке, поскольку тщательный контроль доступа выполняется в фоновом режиме и обычные средства, например, отладчик или пакет **DBMS_OUTPUT,** не помогают. Пакеты, вроде упомянутого в разделе "Ошибки, которые могут возникнуть" пакета **debug,** упрощают трассировку и отладку приложений, использующих средства тщательного контроля доступа.

22

Многоуровневая аутентификация

Многоуровневой (n-tier), или *промежуточной, аутентификацией* называется регистрация программного обеспечения промежуточного уровня от своего имени для выполнения в базе данных каких-либо действий по поручению пользователя. Это позволяет создавать промежуточные приложения, использующие собственную схему аутентификации, например, с помощью сертификатов X509 или другого процесса однократной регистрации и регистрироваться от имени пользователя, не зная его пароля в базе данных. Хотя регистрация выполнена не от имени пользователя, а от имени промежуточного ПО, для базы данных он зарегистрирован.

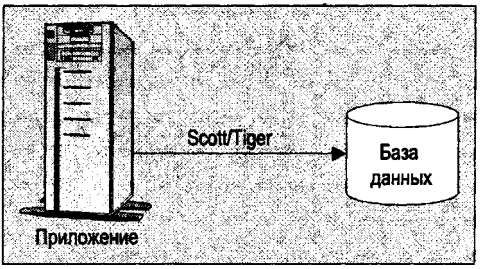
В этой главе мы рассмотрим многоуровневую аутентификацию и использование этой новой возможности Oracle 8i в приложениях. В частности:

- будут представлены возможности и рассмотрено их использование в приложениях;
- разработана программа на основе средств библиотеки OCI, которая позволит использовать промежуточную аутентификацию для регистрации в базе данных;
- описан оператор **ALTER USER**, позволяющий использовать промежуточную аутентификацию в базе данных;
- рассмотрена возможность отслеживания действий, выполняемых от имени промежуточной учетной записи.

В версии Oracle 8i многоуровневую аутентификацию можно использовать только в программах на основе библиотеки Oracle Call interface (OCI), написанных на языках C или C++. В Oracle 9i многоуровневая аутентификация будет поддерживаться также интерфейсом JDBC, что существенно расширит ее возможности.

Когда использовать многоуровневую аутентификацию?

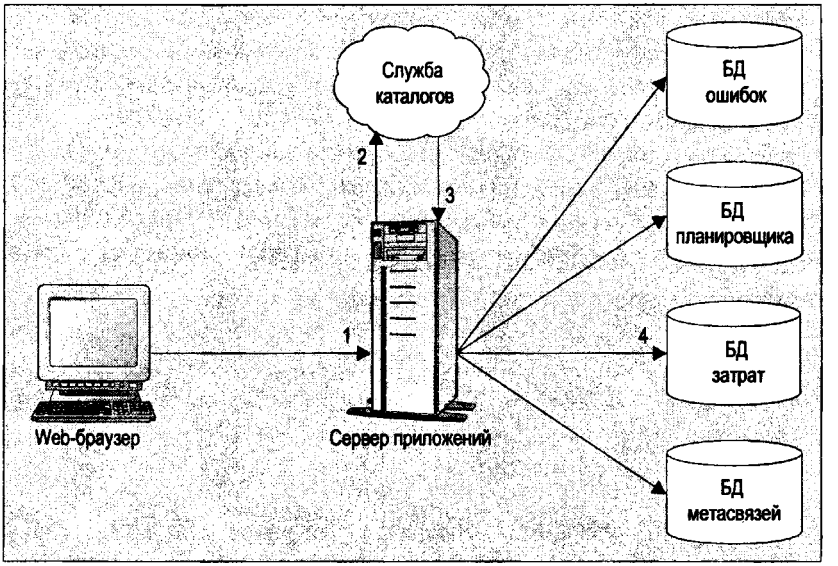
Во времена централизованных и клиент-серверных систем аутентификация была простой. Клиент (приложение) запрашивал у пользователя реквизиты (имя пользователя и пароль) и передавал их серверу базы данных. Сервер проверял эти реквизиты и, если все было правильно, клиент подключался к серверу:



Теперь у нас есть Web и, как следствие, многоуровневая архитектура, когда, например, клиент (браузер) предоставляет свои реквизиты промежуточному серверу приложений, на котором выполняются компоненты JavaServer Page (JSP), обращающиеся к CORBA-объекту, который, в свою очередь, взаимодействует с базой данных. Реквизиты, передаваемые ПО промежуточного уровня, могут совпадать или не совпадать с именем пользователя и паролем, использовавшимся во времена клиент-серверной архитектуры, — это могут быть реквизиты, разрешающие доступ к службе каталогов (directory service), чтобы ПО промежуточного уровня могло выяснить, кто подключается и какими привилегиями доступа. Это могут быть реквизиты в виде сертификата X.509, включающего идентификационную информацию и привилегии. В любом случае ПО промежуточного уровня необязательно использует эти реквизиты для регистрации пользователя в базе данных.

Клиент больше не взаимодействует с базой данных непосредственно; между ними есть один, два или более промежуточных уровней. Можно, правда, заставить пользователя передавать регистрационное имя и пароль компонентам JSP, которые будут передавать их CORBA-объекту, а тот — серверу базы данных, но это не позволит использовать другие технологии и механизмы аутентификации, в особенности механизмы однократной регистрации.

Рассмотрим следующий пример — достаточно типичное современное Web-приложение:



Клиент представляет собой Web-браузер, отображающий HTML-страницы и обращающийся к Web-серверу/серверу приложений по протоколу HTTP (1). Само приложение находится на Web-сервере/сервере приложений, например, в виде Java-сервлета, модуля сервера Apache и т.д. На представленной выше схеме промежуточный сервер использует службу каталогов (directory service), доступную по протоколу LDAP, которой и передаются предоставленные пользователем реквизиты (2). Служба каталогов используется как средство аутентификации сеанса браузера. Если аутентификация прошла успешно, сервер приложений получает соответствующее уведомление (3) и, наконец, подключается к одной из нескольких баз данных (4) для получения данных и обработки транзакций.

"Реквизиты", передаваемые из браузера серверу приложений на шаге (1), могут быть разного вида — имя пользователя/пароль, ключ с сервера однократной регистрации, некий цифровой сертификат — все; что угодно. Имя пользователя базы данных и его пароль, как правило, не передается.

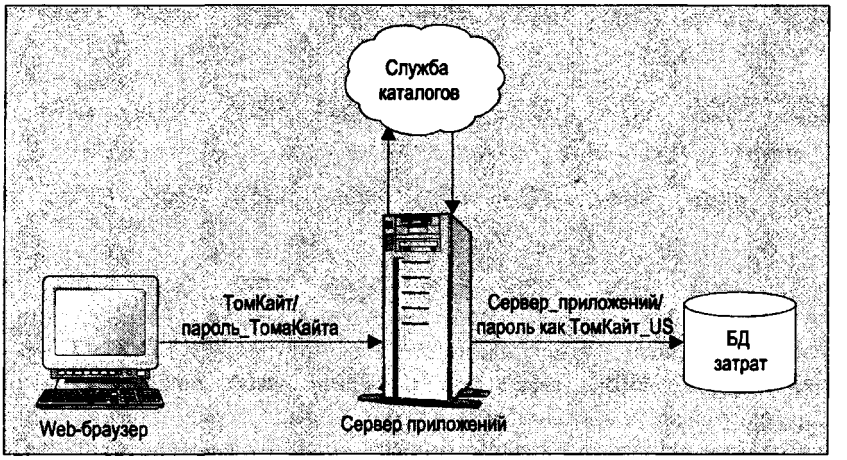
Проблема, конечно же, в том, что серверу приложений необходимо имя пользователя базы данных и пароль для аутентификации пользователя в используемой базе данных. Более того, комбинации имя пользователя/пароль в каждом случае будут разными. В рассмотренном выше примере имеется четыре базы данных:

- база данных ошибок, в которой надо регистрироваться, скажем, как **TKYTE**;
- база данных затрат, в которой надо регистрироваться как **TKYTE_US**;
- база данных планировщика, в которой надо регистрироваться как **WEB$TKYTE**;
- и так далее...

Подумайте: сколько у вас имен пользователей и паролей в разных базах данных? Я могу вспомнить не меньше 15. Более того, хотя имена пользователей в базах данных я никогда не меняю, пароли изменяются достаточно часто. Правда, хорошо было бы аутен-

тифицироваться один раз — на сервере приложений — и обеспечить доступ сервера приложений ко всем необходимым базам данных от нашего имени (или по нашему заданию), не передавая пароли для каждой базы данных? Именно это и обеспечивает многоуровневая аутентификация.

На уровне базы данных для этого достаточно задать опцию подключения. В Oracle 8i оператор **ALTER USER** был изменен и поддерживает конструкцию **GRANT CONNECT THROUGH** (подробнее она рассматривается далее, в разделе "Предоставление привилегии"). Рассмотрим доступ к базе данных затрат в представленном ранее приложении:



Служба каталогов содержит информацию сопоставления, связывающую пользователя **TomKyte** с клиентом базы данных **TKYTE_US**. После успешного получения этой информации сервер приложений (промежуточный сервер) может подключаться к базе данных со своими реквизитами от имени клиента базы данных, **TKYTE_US**. Серверу приложений при этом пароль пользователя **TKYTE_US** знать не надо.

Чтобы это можно было сделать, администратор базы данных затрат должен предоставить учетной записи **APP_SERVER** право подключаться от имени клиента:

```
alter user tykte_us grant connect through app_server
```

Сервер приложений будет работать в базе данных от имени и с привилегиями пользователя **TKYTE_US**, как если бы пользователь **TKYTE_US** зарегистрировался в базе данных непосредственно.

Таким образом, сервер Oracle 8i расширяет модель защиты настолько, что сервер приложений может безопасно работать от имени клиента, не требуя от него пароля соответствующей учетной записи в базе данных и не запрашивая многочисленные привилегии для доступа к объектам или процедурам, которые ему непосредственно не нужны. Кроме того, система проверки также была расширена и включает действия, выполняемые сервером приложений от имени клиента. Другими словами, мы можем узнать, выполнил ли сервер приложений определенное действие от имени клиента (подробнее об этом см. в разделе "Проверка промежуточных учетных записей").

Теперь перейдем к обсуждению реализации этих возможностей. Как упоминалось во введении, это средство в настоящее время* доступно только для программ на языках C и C++, использующих средства Oracle Call Interface.

Механизм многоуровневой аутентификации

В этом разделе одна из стандартных демонстрационных программ OCI, реализующая мини-SQL*Plus, будет изменена для использования промежуточной аутентификации при подключении к базе данных. В результате получится небольшое, интерактивное средство выполнения SQL-операторов, которое позволит выяснить, как работает многоуровневая аутентификация и какие побочные эффекты при этом возникают. В качестве приза вы получите инструментальное средство, которое при наличии соответствующих привилегий позволит зарегистрироваться от имени другого пользователя и выполнять действия, регистрируемые системой проверки как ваши. Это средство можно, например, использовать для предоставления привилегии **SELECT** на таблицу другого пользователя, не зная его пароля.

В демонстрационную программу **cdemo2.c** (которая находится в каталоге **[ORACLE_HOME]\rdbms\demo)** придется добавить только другую процедуру регистрации. После этого мы сможем зарегистрироваться как пользователь **SCOTT,** используя аутентификацию операционной системой, и предоставить роль **CONNECT:**

```
C:\> cdemo2 / scott CONNECT
```

Можно также, например, зарегистрироваться, указав имя пользователя и пароль в удаленной по отношению к учетной записи SCOTT базе данных, и предоставить роли RESOURCE и PLUSTRACE:

```
C:\> cdemo2 user/pass@database scott RESOURCE,PLUSTRACE
```

Для того чтобы показать, как регистрироваться с помощью механизма многоуровневой аутентификации, придется создать C-функцию — эту часть программы мы рассмотрим детально. Остальная же часть приложения — обычный OCI-код, ничем не отличающийся от любой программы, использующей библиотеку OCI.

В начало кода включен стандартный заголовочный файл **oci.h,** находящийся в каталоге **ORACLE_HOME]\rdbms\demo.** Этот файл содержит необходимые прототипы функций и макросов для всех OCI-программ. Затем объявляются локальные переменные для подпрограммы регистрации. Используются обычные дескрипторы подключений OCI, но обратите внимание на два дескриптора **OCISession:** один — для учетной записи, реквизиты которой будут передаваться (от имени которой выполняется регистрация), а второй — для учетной записи, от имени которой мы будем работать. Назначение остальных локальных переменных понятно из имен — они содержат имя пользователя, пароль, имя базы данных и все роли, которые мы хотим предоставить:

```
#include <oci.h>
void checkerr(OCIError * errhp, sword status);
```

```
Lda_Def connect8i(int argc, char * argv[])
{
OCIEnv     *environment_handle;
OCIServer  *data_server_handle;
OCIError   *error_handle;
OCISvcCtx  *application_server_service_handle;
OCISession *first_client_session_handle;
OCISession *application_server_session_handle;

char       *username;
char       *password;
char       *database;
char        temp[255];
char        role_buffer[1024];
char       *roles[255];
int         nroles;
```

Проверим допустимость переданных функции аргументов командной строки. Если передано не четыре аргумента, значит, переданной информации недостаточно, т.е. просто выдается сообщение о правильном использовании, и работа завершается. В противном случае мы анализируем (с помощью стандартной С-функции strtok) переданные аргументы. Поскольку вызов strtok — деструктивный (он изменяет переданную функции строку), перед анализом аргументов мы копируем их в локальные переменные:

```
if (argc != 4)
{
  printf("usage: %s proxy_user/proxy_pass real_account_name rolel,|\n",
          argv[0]);
  printf("          proxy_user/proxy_pass can just be /\n");
  printf("         real_account_name is what you want to connect to\n");
  exit(l);
}
strcpy(temp, argv[1]);
username = strtok(temp, " / " ) ;
password = strtok(NULL, "@");
database=strtok(NULL," " ) ;
strcpy( role_buffer, argv[3] );
for (nroles = 0, roles[nroles] = strtok(role_buffer,",");
     roles[nroles] != NULL;
     nroles++, roles[nroles] =strtok(NULL,","));
```

Теперь выполняем общую инициализацию и выделение контекстов. Это стандартные действия для всех OCI-программ:

```
  OCIInitialize(OCI_DEFAULT, NOLL, NULL, NULL, NULL);

  OCIEnvInit(&environment_handle, OCI_DEFAULT, 0, NULL);

  OCIHandleAlloc((dvoid *) enviroranent_handle,
                 (dvoid **) serror_handle,
                OCI_HTYPE_ERROR, 0, NULL);
```

Затем выделяем и инициализируем контексты сервера и службы, используемые "сервером приложений". В данном случае сервером приложений является демонстрацион-

ная программа cdemo2 — реализация SQL*Plus в миниатюре. Этот код выполняет подключение к серверу, но не начинает сеанс:

```
checkerr(error_handle,
    OCIHandleAlloc(environment_handle,
                   (dvoid **)sdata_server_handIe,
                   OCI_HTYPE_SERVER,
                   0, NULL)
        );
checkerr(error_handle,
    OCIHandleAlloc((dvoid*) environment_handle,
                   (dvoid **) sapplication_server_service_handle,
                   OCI_HTYPE_SVCCTX, 0, NULL)
        );
checkerr(error_handle,
    OCIServerAttach(data_server_handle,
                    error_handle,
                    (text *)database?database:"",
                    strlen(database?database:""), 0)
        );
checkerr(error_handle,
    OCIAttrSet((dvoid *) application_server_service_handle,
               OCI_HTYPE_SVCCTX,
               (dvoid *) data_server_handle,
               (ub4) 0,
               OCI_ATTR_SERVER,
               error_handle)
        );
```

Теперь можно инициализировать, а затем аутентифицировать дескриптор сеанса сервера приложений. В данном случае используется либо внешняя аутентификация, либо имя пользователя/пароль:

```
checkerr(error_handle,
    OCIHandleAlloc(<dvoid *) environment_handle,
                   (dvoid **)&application_server_session_handle,
                   (ub4) OCI_HTYPE_SESSION,
                   (size_t) 0,
                   (dvoid **) 0)
        );
```

Инициализировав дескриптор сеанса, мы должны передать информацию для аутентификации. Можно разрешить либо аутентификацию операционной системой (которая не потребует от сервера приложений передавать имена пользователей и пароли на сервер), либо стандартную аутентификацию по имени пользователя и паролю. Вот код для аутентификации по имени пользователя и паролю:

```
if (username != NULL && password != NULL && *username && *password)
    <
        checkerr(error_handle,
            OCIAttrSet((dvoid *) application_server_session_handle,
```

```
                        (ub4) OCI_HTYPE_SESSION,
                        (dvoid *) username, (ub4) strlen((char *)username),
                        (ub4) OCI_ATTR_USERNAME, error_handle)
                );
        checkerr( error_handle,
            OCIAttrSet((dvoid *) application_server_session_handle,
                        (ub4) OCI_HTYPE_SESSION,
                        (dvoid *) password,
                        (ub4) strlen((char *)password),
                        (Ub4) OCI_ATTR_PASSWORD,
                       error_handle)
                );
        checkerr(error_handle,
            OCISessionBegin (application_server_service_handle,
                             error_handle,
                             application_server_session_handle,
                             OCI_CRED_RDBMS,
                               (ub4) OCI_DEFAULT)
                );
    }
```

А это — для выполнения аутентификации операционной системой:

```
    else
    {
        checkerr(error_handdle,
                 OCISessionBegin(application_server_service_handJ_e,
                                 error_handle,
                                 application_server_session_handle,
                                 OCI_CRED_EXT,
                                 OCI_DEFAULT)
                );
    }
```

Теперь все готово для инициализации сеанса от имени клиента (пользователя, который регистрируется на сервере приложений, и от имени которого нам доверяют выполнять действия). Сначала инициализируем сеанс:

```
checkerr (error_handle,
          OCIHandleAlloc((dvoid *) environment_handle,
                         (dvoid **)&first_client_session_handle,
                         (ub4)  OCI_HTYPE_SESSION,
                         (size_t) 0,
                         (dvoid **) 0)
         );
```

Потом устанавливаем имя пользователя, от имени которого этот сеанс будет работать:

```
    checkerr(error_handle,
         OCIAttrSet((dvoid *) first_client_session_handle,
                    (ub4) OCI_HTYPE_SESSION,
                    (dvoid *) argv[2],
```

```
                    (ub4) strlen(argv[2]),
                    OCI_ATTR_USERNAME,
                    error_handle)
          );
```

Затем добавляем список ролей, которые необходимо включить для данного сеанса, — если пропустить этот вызов, будут включены все стандартные роли соответствующего пользователя:

```
checkerr( error_handle,
     OCIAttrSet((dvoid *)  first_client_session_handle,
                (ub4) OCI_HTYPE_SESSION,
                (dvoid *) roles,
                (ub4) nroles,
                OCI_ATTR_INITIAL_CLIENT_ROLES,
                error_handle)
        );
```

Теперь все готово для начала реального сеанса. Сначала мы свяжем клиентский сеанс (от имени которого должны выполняться действия в базе данных) с сеансом сервера приложений (промежуточной учетной записью):

```
checkerr(error_handle,
         OCIAttrSet((dvoid *)  first_client_session_handle,
                    (ub4) OCI_HTYPE_SESSION,
                    (dvoid *)  application_server_session_handle,
                    (ub4) 0,
                    CCI_ATTR_PROXY_CREDENTIALS,
                    error_handle)
         );

checkerr(error_handle,
      OCIAttrSet((dvoid  *)application_server_service_handle,
                 (ub4) OCI_HTYPE_SVCCTX,
                 (dvoid  *)first_client_session_handle,
                 (ub4)0,
                 (ub4)OCI_ATTR_SESSION,
                 error_handle)
        );
```

А затем начнем сеанс:

```
checkerr(error_handle,
        OCISessionBegin(application_server_service_handle,
                        error_handle,
                        first_client_session_handle,
                        OCI_CRED_PROXY,
                        OCI_DEFAULT)
         );
```

Теперь, поскольку это OCI-программа версии 7 **(cdemo2.c** — программа версии 7), необходимо преобразовать регистрационные данные Oracle 8i в форму, которую можно будет использовать. Здесь будет преобразована информация о подключении версии 8 в OCI LDA (Login Data Area — область данных регистрации) версии 7 и возвращен результат:

```
checkerr(error_handle,
         OCISvcCtxToLda(application_server_service_handle,
                        error_handle,
                        &lda )
         );
    return lda;
}
```

Последняя часть кода — функция checkerr, которую мы многократно использовали. Эта функция проверяет, что коды возврата OCI-функций свидетельствуют об успешном выполнении — в противном случае она выдает сообщение об ошибке и завершает работу программы:

```
void checkerr(OCIError * errhp, sword status)
{
text        errbuf[512];
sb4         errcode = 0;

  switch (status)
  {
  case OCI_SUCCESS:
    break;
  case OCI_SUCCESS_WITH_INFO:
     (void) printf("Ошибка - OCI_SUCCESS_WITH_INFO\n");
    break;
  case OCI_NEED_DATA:
    (void)printf("Ошибка- OCI_NEED_DATA\n");
    break;
  case OCI_NO_DATA:.
    (void)printf("Ошибка- OCI_NODATA\n");
    break;
  case OCI_ERROR:
    (void) OCIErrorGet((dvoid *)errhp, (ub4) 1, (text *) NULL, &errcode,
                       errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
    (void) printf("Ошибка-%.*s\n", 512, errbuf);
    exit(1);
    break;
  case OCI_INVALID_HANDLE:
     (void) printf("Ошибка - OCI_INVALID_HANDLE\n");
    break;
  case OCI_STILL_EXECUTING:
    (void)printf("Ошибка- OCI_STILL_EXECUTE\n");
    break;
  case OCI_CONTINUE:
     (void) printf("Ошибка - OCI_CONTINUE\n");
    break;
  default:
    break;
  }
}
```

Теперь осталось изменить файл **cdemo2.c** и включить в него необходимый код. Существующий код этой демонстрационной программы имеет вид:

```
static sword numwidth = 8;
main()
{
    sword col, errno, n, ncols;
    text *cp;
    /* Подключаемся к ORACLE. */
    if (connect_user())
        exit(-1);
```

Изменение очень несложное — добавить код, выделенный полужирным:

```
static sword numwidth = 8;
Lda_Def connect8i(int argc, char * argv[]);
main(int argc, char * argv[])
{
    sword col, errno, n, ncols;
    text *cp;
    /* Подключаемся к ORACLE. */
    /*
    if (connect_user())
        exit(-1);
    */
    Ida = connect8i( argc/ argv );
```

Затем надо добавить весь представленный ранее код (функции connect8i и checkerr) в конец файла исходного кода. Сохраняем файл и компилируем.

В ОС UNIX для компиляции надо выполнить следующую команду:

```
$ make -f $ORACLE_HOME/rdbms/demo/demo_rdbms.mk cdemo2
```

В среде Windows NT я использовал следующий файл управления проектом makefile:

```
CPU=i386
WORK_DIR        = .\
!include <\msdev\include\win32.mak>
OBJDIR    = $(WORK_DIR)\    #
EXEDIR =  $(WORK_DIR)\    #  каталог, в который будут помещаться все
➥  .exe-модули
ORACLE_HOME = \oracle
TARGET   = $(EXEDIR)cdemo2.exe

SAMPLEOBJS   = cdemo2.obj
LOCAL_DEFINE = -DWIN_NT
SYSLIBS = \msdev\lib\msvcrt.lib \
          \msdev\lib\oldnames.lib \
          \msdev\lib\kernel32.lib  \
```

```
          \msdev\lib\advapi32.lib  \
          \msdev\lib\wsock32.lib

NTUSER32LIBS  = \msdev\lib\user32.1ib \
                \msdev\lib\advapi32.1ib \
                \msdev\lib\libc.lib
SQLLIB  = $(ORACLE_HOME)\oci\lib\msvc\oci.lib

INCLS   = -I\msdev\include \
          -I$(ORACLE_HOME)\oci\include
CFLAGS  = $(cdebug) $(cflags) $(INCLS) $(LOCAL_DEFINE)
LINKOPT = /nologo /subsystem:console /machine:I386 /nodefaultlib

$(TARGET): $(SAMPLEOBJS) $(SQLLIB)
    $(link) $(LINKOPT) \
    -out:$(TARGET) $(SAMPLEOBJS) \
        $(NTUSER32LIBS) \
        $(SYSLIBS) \
        $(SQLLIB)
```

А затем просто выполнял для компиляции команду nmake:

```
c:\oracle\rdbms\demo>nmake
```

Теперь можно проверять работу программы:

```
c:\oracle\rdbms\demo>cdemo2 tkyte/tkyte scott connect,resource
Ошибка — ORA-28150: proxy not authorized to connect as client
```

Итак, пока еще не работает, но до успешного выполнения осталось совсем немного. Необходимо предоставить промежуточной учетной записи (**TKYTE**) право подключаться от имени клиента базы данных (**SCOTT**). Регистрируемся в SQL*Plus и выполняем:

```
sys@TKYTE816> alter user scott grant connect through tkyte;
User altered.
```

Подробно этот новый оператор со всеми опциями мы рассмотрим немного позже. Пока нам просто надо убедиться, что программа работает. Обратите внимание на выделенное **полужирным** приглашение — я работаю не в среде SQL*Plus. Это работает демонстрационная программа **cdemo2,** очень похожая на утилиту SQL*Plus:

```
c:\oracle\rdbms\demo>cdemo2 tkyte/tkyte scott connect,resource

OCISQL> SELECT user, substr(sys_context('userenv','proxy_user'),1,30)
  2     FROM dual;

USER                           SUBSTR(SYS_CONTEXT('USERENV','PRO

SCOTT                          TKYTE

1 row processed.

OCISQL> select * from session_roles;

ROLE

CONNECT
RESOURCE

2 rows processed.
```

```
OCISQL> select distinct authentication_type from v$session_connect_info
  2       where sid = (select sid from v$mystat where rownum =1);

AUTHENTICATION_TYPE

PROXY

1 row processed.

OCISQL> exit

C:\oracle\RDBMS\demo>
```

Вот и все. Мы успешно зарегистрировались от имени пользователя **SCOTT,** не зная его пароля. Кроме того, мы увидели, как можно проверить, что мы работаем через промежуточную учетную запись, сравнивая значение **USER** с атрибутом **PROXY_USER,** возвращаемым функцией **SYS_CONTEXT**, или просматривая информацию сеанса в представлении **V$SESSION_CONNECT_INFO**. Кроме того, явно видно, что включены роли **CONNECT** и **RESOURCE**. Если подключиться следующим образом:

```
c:\oracle\rdbms\demo>cdemo2 tkyte/tkyte scott connect

OCISQL> select * from session_roles;

ROLE

CONNECT

1 row processed.
```

Видно, что включена роль **CONNECT**, — мы контролируем, какие роли включает "сервер приложений".

Предоставление привилегии

Соответствующий оператор **ALTER USER** имеет следующий базовый синтаксис:

```
Alter user <имя пользователя> grant connect through
<промежуточный пользователь><, промежуточный пользователь>...
```

Это дает возможность пользователям, перечисленным в списке промежуточных пользователей, подключаться от имени указанного после **ALTER USER** пользователя. По умолчанию для этих пользователей будут устанавливаться все роли данного пользователя. Другая разновидность этого оператора:

```
Alter user <имя пользователя> grant connect through
<промежуточный пользователь> WITH NONE;
```

позволяет промежуточной учетной записи подключаться от имени указанного пользователя, но только с ее базовыми привилегиями —роли включаться не будут. Кроме того, можно использовать:

```
Alter user <имя пользователя> grant connect through
<промежуточный пользователь> ROLE имя_роли,имя_роли,...
```

или:

```
Alter user <имя пользователя> grant connect through
<промежуточный пользователь> ROLE ALL EXCEPT имя_роли,имя_роли,...
```

Два представленных выше оператора дают промежуточной учетной записи возможность подключаться в качестве пользователя, но при этом включены будут только определенные роли. Необязательно давать учетной записи сервера приложений все привилегии — достаточно предоставить роли, необходимые для выполнения его функций. По умолчанию сервер Oracle пытается включить все стандартные роли пользователя и роли **PUBLIC**. Вполне допустимо разрешить серверу приложений использовать только роль HR данного пользователя, и никакие другие прикладные роли этого пользователя.

Разумеется, можно и отобрать соответствующую привилегию:

```
Alter user <имя пользователя> REVOKE connect through <промежуточный
пользователь><,промежуточный пользователь>...
```

Есть административное представление, **PROXY_USERS**, которое можно использовать для получения информации обо всех промежуточных учетных записях. После выполнения оператора **ALTER** user **SCOTT GRANT CONNECT** through tkyte в представлении **PROXY_USERS** будет:

```
TKYTE@TKYTE816> select * from proxy_users;

PROXY    CLIENT   ROLE       FLAGS

TKYTE    SCOTT               PROXY MAY ACTIVATE ALL CLIENT ROLES
```

Проверка промежуточных учетных записей

Вот новый синтаксис оператора AUDIT для работы с промежуточными учетными записями:

```
AUDIT <действие> BY <промежуточный пользователь>,
<промежуточный пользователь> ... ON BEHALF OF <клиент>, <клиент>.. ;
```

или:

```
AUDIT <действие> BY <промежуточный пользователь>,
<промежуточный пользователь> ON BEHALF OF ANY;
```

Новая часть — **BY <промежуточный пользователь>... ON BEHALF OF.** Она позволяет явно проверять действия, выполняемые указанными промежуточными пользователями от имени некоторых или всех учетных записей.

Предположим, администратор включил проверку, установив параметр инициализации **AUDIT_TRAIL=TRUE** и перезапустив экземпляр. Тогда можно использовать:

```
sys@TKYTE816> audit connect by tkyte on behalf of scott;
Audit succeeded.
```

Теперь, если использовать для подключения измененную программу cdemo2.c:

```
C:\oracle\RDBMS\demo>cdemo2 tkyte/tkyte scott connect

OCISQL> exit
```

В таблице **DBA_AUDIT_TRAIL** я обнаружу следующее:

```
OS_USERNAME                    : Thomas?Kyte
USERNAME                       : SCOTT
USERHOST                       :
TERMINAL                       : TKYTE-DELL
TIMESTAMP                      : 08-may-2001 19:19:29
OWNER                          :
OBJ_NAME                       :
ACTION                         : 101
ACTION_NAME                    : LOGOFF
NEW_OWNER                      :
NEW_NAME                       :
OBJ_PRIVILEGE                  :
SYS_PRIVILEGE                  :
ADMIN_OPTION                   :
GRANTEE                        :
AUDIT_OPTION                   :
SES_ACTIONS                    :
LOGOFF_TIME                    : 08-may-2001 19:19:30
LOGOFF_LREAD                   : 23
LOGOFF_PREAD                   : 0
LOGOFF_LWRITE                  : 6
LOGOFF_DLOCK                   : 0
COMMENT_TEXT                   : Authenticated by: PROXY: TKYTE
SESSIONID                      : 8234
ENTRYID                        : 1
STATEMENTID                    : 1
RETORNCODE                     : 0
PRIV USED                      : CREATE SESSION
```

Интересно отметить, что при подключении пользователя **SCOTT** или пользователя TKYTE с помощью утилиты SQL*Plus записи проверки не создаются. Проверка строго ограничена конструкцией:

```
connect by tkyte on behalf of scott;
```

При необходимости можно проверять подключение пользователей **TKYTE** или **SCOTT**, просто в данном случае я решил этого не делать. Этот пример демонстрирует, что средства многоуровневой аутентификации не противоречат учету действий в базе данных (можно определить, что именно пользователь **SCOTT** выполнил определенное действие), позволяя также определить, когда действие от имени пользователя **SCOTT** выполнил сервер приложений.

Проблемы

Обычно многоуровневая аутентификация работает вполне предсказуемо. Если подключиться следующим образом:

```
C:\oracle\RDBMS\demo>cdemo2 tkyte/tkyte scott connect
```

Это будет равносильно непосредственной регистрации пользователя **SCOTT**. Средства обеспечения работы с правами вызывающего и правами создателя (см. главу 23) функционируют так, как если бы зарегистрировался пользователь **SCOTT**. Средства тща-

тельного контроля доступа (см. главу 21) функционируют так, как если бы зарегистрировался пользователь **SCOTT**. Триггеры на событие регистрации пользователя **SCOTT** срабатывают. И так далее. Я не нашел ни одной возможности, которая была бы недоступна при использовании многоуровневой аутентификации.

Есть, однако, одна деталь реализации, которая может привести к проблемам. При использовании многоуровневой аутентификации сервер будет автоматически включать набор ролей. Если вы используете атрибут **OCI_ATTR_INITIAL_CLIENT_ROLES**, как в представленном выше коде, то ожидаете, что в набор войдут только явно заданные роли. Однако всегда включаются также все роли, предоставленные роли **PUBLIC**. Например, если предоставить роль **PLUSTRACE** роли **PUBLIC** (роль **PLUSTRACE** позволяла использовать установку **AUTOTRACE**, которую мы постоянно применяли по ходу изложения в среде SQL*Plus для оценки производительности):

```
sys@TKYTE816>grant plustrace to public-
Grant succeeded.

Теперь при подключении с помощью нашей утилиты мини-SQL*Plus:

c:\oracle\rdbms\demo>cdemo2 tkyte/tkyte scott connect

OCISQL> select * from session_roles;

ROLE


CONNECT
PLUSTRACE

2 rows processed.
```

оказывается, что кроме роли CONNECT включена также роль PLUSTRACE. Сначала это не кажется опасным. Однако, если с помощью оператора ALTER USER явно потребовать предоставлять пользователю только строго ограниченный набор ролей:

```
sys@TKYTE816>alter user scott grant connect through tkyte with role
-> CONNECT;
User altered.
```

окажется, что:

```
c:\oracle\rdbms\demo>cdemo2 tkyte/tkyte scott connect
Ошибка - ORA-28156: Proxy user 'TKYTE' not authorized to set role
 'PLUSTRACE' for client 'SCOTT'
```

пользователю **TKYTE** не разрешено включать эту роль при подключении от имени пользователя **SCOTT**. Решить эту проблему можно только так:

1. не предоставлять ролей роли **PUBLIC**;
2. всегда добавлять соответствующие роли к списку ролей в операторе **ALTER USER**

Например, если выполнить:

```
sys@TKYTE816>alter user scott grant connect through tkyte with role
  2 connect, plustrace;

User altered.
```

следующая команда сработает, как и предполагалось:

```
c:\oracle\rdbms\demo>cdemo2 tkyte/tkyte scott connect
OCISQL> select * from session_roles;
ROLE

CONNECT
PLUSTRACE
```

Резюме

В этой главе мы изучили возможности многоуровневой, или промежуточной, аутентификации, которые доступны при программировании с использованием библиотеки OCI. Многоуровневая аутентификация позволяет серверу приложений промежуточного уровня действовать в качестве пользующегося доверием агента в базе данных от имени известного приложению клиента. Мы рассмотрели, как сервер Oracle позволяет ограничить набор ролей, доступных промежуточной учетной записи сервера приложений, чтобы от имени промежуточной учетной записи можно было выполнять только ограниченный набор действий, необходимых приложению. Далее мы рассмотрели систему проверки, которая теперь поддерживает использование многоуровневой аутентификации. Можно регистрировать действия, выполняемые промежуточными учетными записями от имени любого указанного пользователя или от имени всех пользователей. При этом всегда легко определить, когда данный пользователь выполнил действие через промежуточную учетную запись, а когда — непосредственно.

Изменив одну демонстрационную программу Oracle, мы получили простую среду, напоминающую SQL*Plus, для тестирования возможностей многоуровневой аутентификации. Эта среда идеально подходит для тестирования различных особенностей многоуровневой аутентификации и позволяет изучать их в интерактивном режиме.

23

Права вызывающего и создателя

Для начала представим ряд определений, чтобы гарантировать однозначное понимание терминов *вызывающий* и *создатель:*

- **Создатель**. Пользователь, создавший *скомпилированный хранимый объект* и владеющий им. (Говорят также, что объект находится *в схеме* пользователя.) К скомпилированным хранимым объектам относятся пакеты, процедуры, функции, триггеры и представления,
- **Вызывающий.** Пользователь, с привилегиями которого работает текущий сеанс. Это может быть текущий зарегистрированный пользователь, но может быть и другой пользователь.

До версии Oracle 8i все скомпилированные хранимые объекты выполнялись с правами создателя объекта. По отношению к соответствующей схеме происходило и разрешение имен. Другими словами, набор привилегий, непосредственно предоставленных владельцу (создателю) объекта использовался при компиляции для определения того:

- к каким объектам (таблицам и т.д.) фактически обращаться;
- есть ли у создателя необходимые для доступа к этим объектам привилегии.

Это статическое связывание, выполняемое на этапе компиляции, распространяется так далеко, что учитываются только непосредственно предоставленные создателю привилегии (другими словами, в ходе компиляции и выполнения хранимой процедуры роли не учитываются). Кроме того, при выполнении процедуры с правами создателя, она будет работать с базовым набором привилегий создателя, а не вызвавшего процедуру.

Начиная с Oracle 8i появилась возможность выполнять процедуры с правами вызывающего, что позволяет создавать процедуры, функции и пакеты, выполняющиеся с набором привилегий вызывающего, а не создателя. В этой главе мы рассмотрим:

- когда следует использовать процедуры с правами вызывающего, в том числе, для создания приложений, работающих со словарем данных, универсальных объектных типов и реализации собственных средств контроля доступа;
- когда следует использовать процедуры с правами создателя, с учетом их масштабируемости по сравнению с процедурами, работающими с правами вызывающего, а также возможностей по реализации защиты данных;
- как работают хранимые процедуры этих двух типов;
- проблемы, которые необходимо учитывать при выполнении процедур с правами вызывающего, в частности использование разделяемого пула, производительность процедур, необходимость расширенных средств обработки ошибок в коде, а также использование языка Java для реализации процедур, работающих с правами вызывающего.

Пример

Возможность выполнять код с правами вызывающего позволяет, например, создать хранимую процедуру, работающую с набором привилегий пользователя, который ее выполняет. В результате хранимая процедура может работать правильно и корректно для одного пользователя (который имеет доступ ко всем необходимым объектам), но не работать для другого (у которого такого доступа нет). Причина состоит в том, что доступ к базовым объектам проверяется не во время компиляции, а во время выполнения (правда, создатель должен иметь доступ к соответствующим объектам или хотя бы к объектам с такими же именами, чтобы PL/SQL-код вообще можно было скомпилировать). Подобный доступ во время выполнения осуществляется с учетом привилегий и ролей текущего пользователя/схемы. Следует отметить, что работа с правами вызывающего не поддерживается для представлений и триггеров. Представления и триггеры создаются и работают только с правами создателя.

Возможность работы с правами вызывающего легко реализовать и проверить, поскольку для этого необходимо добавить к процедуре или пакету всего одну строку. Рассмотрим, например, следующую процедуру, выдающую значения атрибутов контекста:

- **CURRENT_USER**. Имя пользователя, с привилегиями которого работает сеанс.
- **SESSION_USER**. Имя пользователя, зарегистрировавшегося и первоначально создавшего этот сеанс. Это значение в течение сеанса неизменно.
- **CURRENT_SCHEMA**. Имя стандартной схемы, которая будет использоваться при разрешении неуточненных ссылок на объекты.

Для создания процедуры, работающей с правами создателя, необходим следующий код:

```
tkyte@TKYTE816> create or replace procedure definer_proc
  2  as
```

```
  3  begin
  4      for x in
  5      (select sys_context('userenv', 'current_user') current_user,
  6              sys_context('userenv', 'session_user') session_user,
  7         sys_context('userenv',        'current_schema') current_schema
  8           from dual)
  9      loop
 10         dbms_output.put_line('CurrentUser:    ' || x.current_user);
 11         dbms_output.put_line('SessionUser:    ' || x.session_user);
 12     dbms_output.put_line('Current     Schema:  ' || x.current_schema);
 13      end loop;
 14  end;
 15  /

Procedure created.
tkyte@TKYTE816> grant execute on definer_proc to scott;

Grant succeeded.
```

Для создания такой же процедуры, работающей с правами вызывающего, код надо немного изменить:

```
tkyte@TKYTE816> create or replace procedure invoker_proc
  2  AUTHID CURRENT_USER
  3  as
  4  begin
  5      for x in
  6      (select sys_context('userenv', 'current_user') current_user,
  7              sys_context('userenv', 'session_user') session_user,
  8              sys_context('userenv', 'current_schema') current_schema
  9           from dual)
 10      loop
 11     dbms_output.put_line('Current     User:    ' || x.current_user);
 12         dbms_output.put_line('SessionUser:    ' || x.session_user);
 13        dbms_output.put_line('Current Schema: ' || x.current_schema);
 14      end loop;
 15  end;
 16  /

Procedure created.

tkyte@TKYTE816> grant execute on invoker_proc to scott;

Grant succeeded.
```

Вот и все; добавили одну строку, и процедура теперь будет выполняться с привилегиями и в пространстве имен вызывающего пользователя, а не создателя. Чтобы глубже понять, что это означает, выполним представленные выше процедуры и сравним выдаваемые ими результаты. Сначала процедура, работающая с правами создателя:

```
tkyte@TKYTE816> connect scott/tiger

scott@TKYTE816> exec tkyte.definer_proc
Current User:   TKYTE
Session User:   SCOTT
Current Schema: TKYTE

PL/SQL procedure successfully completed.
```

Внутри процедуры, работающей с правами создателя, текущий пользователь и схема, с привилегиями которой работает сеанс — **TKYTE**. Пользователь, зарегистрировавшийся и начавший сеанс — **SCOTT**. Это значение в течение сеанса не меняется. При этом все неуточненные ссылки будут разрешаться в схеме TKYTE (например, запрос **SELECT * FROM T** будет разрешаться как **SELECT * FROM TKYTE.T**).

Процедура, работающая с правами вызывающего, дает совсем другие результаты:

```
scott@TKYTE816>exectkyte.invoker_proc
Current User:  SCOTT
Session User:  SCOTT
Current Schema: SCOTT

PL/SQL procedure successfully completed.
```

Текущий пользователь — **SCOTT**, а не **TKYTE**. Текущий пользователь в такой процедуре совпадает с пользователем, непосредственно выполняющим эту процедуру. Пользователь, зарегистрировавшийся и начавший сеанс — **SCOTT**, как и ожидалось. Текущая схема, однако, — тоже **SCOTT**, поэтому запрос **SELECT * FROM T** будет выполняться как **SELECT * FROM SCOTT**.T. Это показывает фундаментальное отличие процедур, работающих с правами вызывающего: процедура работает с правами пользователя, который ее вызвал. Кроме того, текущая схема также зависит от вызывающего. При выполнении этой процедуры разными пользователями она может обращаться к различным объектам.

Интересно, как повлияет на эти процедуры явное изменение текущей схемы:

```
scott@TKYTE816> alter session set current_schema - system;

Session altered.

scott@TKYTE816>exectkyte.definer_proc
Current User:  TKYTE
Session User:  SCOTT
Current Schema: TKYTE

PL/SQL  procedure successfully completed.

scott@TKYTE816>exectkyte.invoker_proc
Current User:  SCOTT
Session User:  SCOTT
Current Schema: SYSTEM

PL/SQL procedure successfully completed.
```

Как видите, результаты выполнения процедуры с правами создателя не изменились. В таких процедурах текущий пользователь и текущая схема "статичны". Они жестко задаются при компиляции и не меняются в дальнейшем при изменении текущей среды. Процедура, работающая с правами вызывающего, более динамична. Текущий пользователь устанавливается во время выполнения, а текущая схема может меняться для каждого вызова, даже в пределах одного сеанса.

Это очень мощное средство (при правильном и уместном использовании). Оно позволяет реализовать для хранимых процедур и пакетов PL/SQL поведение, более свойственное приложениям, написанным на Pro*C. Приложение, использующее Pro*C(или интерфейсы ODBC, JDBC — в общем, любое клиентское приложение на обычных про-

цедурных языках программирования), выполняется с привилегиями текущего зарегистрированного пользователя (вызывающего) и разрешением имен в его схеме. Теперь можно писать на PL/SQL код, который ранее приходилось писать на обычных языках программирования вне базы данных.

Когда использовать права вызывающего

В этом разделе мы рассмотрим различные причины и случаи, когда может потребоваться выполнение с правами вызывающего. Мы сконцентрируемся на правах вызывающего, поскольку это новая возможность, пока еще являющаяся исключением. Хранимые процедуры ранее всегда выполнялись сервером Oracle с правами создателя.

Необходимость работать с правами вызывающего чаще всего возникает, когда универсальный фрагмент кода создается одним пользователем, а используется — множеством других. Разработчик не имеет доступа к объектам, к которым будут иметь доступ пользователи. Именно привилегии пользователя будут определять, к каким объектам этот код может обращаться. Другая потенциальная причина использования прав вызывающего — необходимость создать набор процедур, централизованно выбирающих данные из нескольких различных схем. При использовании процедур, работающих с правами создателя, как было показано, привилегии и схема, относительно которой выполняется разрешение имен, — статичны и определяются во время компиляции. При каждом выполнении процедура, работающая с правами создателя, обращается к одному и тому же набору объектов (если, конечно, не используется динамическое формирование SQL-операторов). Работа с правами вызывающего позволяет создать процедуру, способную обращаться к аналогичным структурам в различных схемах, — в зависимости от того, кто ее вызвал.

Давайте рассмотрим ряд типичных случаев, когда используются процедуры с правами вызывающего.

Разработка универсальных утилит

Пусть создается хранимая процедура, использующая динамический SQL для выполнения запроса и выдачи результатов в виде файла со значениями через запятую. Если не работать с правами вызывающего, эта процедура будет универсальной и полезной всем только при выполнении одного из следующих условий.

- **Создатель процедуры должен иметь возможность читать любой объект в базе данных.** Например, обладать привилегией **SELECT ANY TABLE. В** противном случае при запуске этой процедуры для представления данных таблицы в виде текстового файла произойдет ошибка, потому что создатель не имеет необходимой привилегии SELECT для этой таблицы. Надо выполнять эту процедуру с правами вызывающего, а не создателя.

- **Каждый пользователь должен иметь исходный код и устанавливать его копию в своей схеме.** Это нежелательно по очевидным причинам — сопровождение превратится в кошмар. Если будет обнаружена ошибка в исходном коде или вследствие обновления версии придется изменять код, обновлять придется десятки

копий. Кроме того, эта скопированная процедура все равно не сможет обращаться к объектам, доступным пользователю через роль.

Обычно именно второй вариант чаще всего применяется для разработки универсального кода. Этот подход неидеален, но более "безопасен" с точки зрения защиты данных. Используя работу с правами вызывающего, можно создать процедуру **один раз,** предоставить права на ее выполнение многим пользователям, и они будут использовать ее со своим набором привилегий и с разрешением имен в своих схемах. Давайте рассмотрим небольшой пример. Мне часто приходится просматривать в среде SQL*Plus слишком "широкие" таблицы, имеющие много столбцов. Если просто выполнить **SELECT * FROM** T для такой таблицы, утилита SQL*Plus будет переносить данные на следующую строку по правому краю окна терминала. Например:

```
tkyte@DEV816> select * from dba_tablespaces where rownum = 1;

TABLESPACE_NAME                INITIAL_EXTENT NEXT_EXTENT MIN_EXTENTS

MAX_EXTENTS PCT_INCREASE MIN_EXTLEN STATUS   CONTENTS  LOGGING

EXTENT_MAN ALLOCATIO PLU

SYSTEM                                  16384      16384           1
       505          50          0 ONLINE    PERMANENT LOGGING
DICTIONARY USER     NO
```

Полученные данные читать очень неудобно. Вот если бы получать результаты в следующем виде:

```
tkyte@DEV816> exec print_table('select * from dba_tablespaces where
-> rownum = 1');

TABLESPACE_NAME                : SYSTEM
INITIAL_EXTENT                 : 16384
NEXT_EXTENT                    : 16384
MIN_EXTENTS                    : 1
MAX_EXTENTS                    : 505
PCT_INCREASE                   : 50
MIN_EXTLEN                     : 0
STATUS                         : ONLINE
CONTENTS                       : PERMANENT
LOGGING                        : LOGGING
EXTENT_MANAGEMENT              : DICTIONARY
ALLOCATION_TYPE                : USER
PLUGGED_IN                     : NO


PL/SQL procedure successfully completed.
```

Да, так гораздо лучше! Легко увидеть значение каждого столбца. Увидев результаты использования моей процедуры **PRINT_TABLE**, все хотят получить ее. Вместо того чтобы давать код, я предлагаю использовать мою, поскольку она создана с конструкцией **AUTHID CURRENT_USER**. Мне не нужен доступ к чужим таблицам. Эта процедура сможет обращаться к ним (даже к тем, которые доступны через роль, — процедуры,

работающие с правами создателя, не могут этого делать в принципе). Давайте рассмотрим код и разберемся, как он устроен. Начнем с создания служебной учетной записи для хранения этого универсального кода, а также учетной записи, которую мы будем использовать для проверки зашиты:

```
tkyte@TKYTE816> grant connect to another_user identified by another_user;

Grant succeeded.

tkyte@TKYTE816> create user utils_acct identified by utils_acct;

User created.

tkyte@TKYTE816> grant create session, create procedure to utils_acct;

Grant succeeded.
```

Я создал пользователя с очень ограниченными привилегиями. Их достаточно для того, чтобы зарегистрироваться и создать процедуру. Теперь я создам процедуру в этой схеме:

```
tkyte@TKYTE816> utils_acct/utils_acct

utils_acct@TKYTE816> create or replace
  2  procedure print_table(p_query in varchar2)
  3  AUTHID CURRENT_USER
  4  is
  5       l_theCursor     integer default dbms_sql.open_cursor;
  6       l_columnValue   varchar2(4000);
  7       l_status        integer;
  8       l_descTbl       dbms_sql.desc_tab;
  9       l_colCnt        number;
 10  begin
 11     dbms_sql.parse(l_theCursor, p_query, dbms_sql.native);
 12     dbms_sql.describe_columns(l_theCursor, l_colCnt, l_descTbl);
 13
 14     for i in 1 .. l_colCnt loop
 15        dbms_sql.define_column(l_theCursor, i, l_columnValue, 4000);
 16     end loop;
 17
 18     l_status :=dbms_sql.execute(l_theCursor);
 19
 20     while (dbms_sql.fetch_rows(l_theCursor) >0) loop
 21         for i in 1 .. l_colCnt loop
 22            dbms_sql.column_value(l_theCursor, i, l_columnValue);
 23            dbms_output.put_line(rpad(l_descTbl(i).col_name, 30)
 24                                 || ': ' ||
 25                                  l_columnValue);
 26         end loop;
 27         dbms_output.put_line('-----------------•) ;
 28     end loop;
 29  exception
 30     when others then
 31          dbms_sql.dose_cursor(l_theCursor);
 32          RAISE;
 33  end;
```

```
  34  /

Procedure created.

utils_acct@TKYTE816> grant execute on print_table to public;

Grant succeeded.
```

Теперь я пойду на шаг дальше: сделаю так, чтобы от имени учетной записи UTILS_ACCT зарегистрироваться вообще было невозможно. Это предотвратит подбор пользователем пароля учетной записи UTILS_ACCT и размещение "троянского" кода под видом процедуры PRINT_TABLE. Конечно, администратор базы данных с соответствующими привилегиями сможет снова активизировать эту учетную запись и зарегистрироваться от имени UTILS_ACCT — этого предотвратить нельзя:

```
utils_acct@TKYTE816>connect tkyte/tkyte

tkyte@TKYTE816>revoke create session, create procedure
  2  from utils_acct;

Revoke succeeded.
```

Итак, имеется учетная запись, в схеме которой хранится код, но она по сути заблокирована, поскольку больше не имеет привилегии **CREATE SESSION.** При регистрации от имени пользователя **SCOTT** оказывается, что мы не только по-прежнему можем использовать эту процедуру (хотя учетная запись **UTILS_ACCT** лишена всех привилегий), но и обращаться к своим таблицам. Убедимся, что другие пользователи не могут использовать эту процедуру для доступа к нашим таблицам (если только они не сделают это непосредственно, с помощью запроса), т.е. продемонстрируем, что процедура работает с привилегиями вызывающего:

```
scott@TKYTE816>exec utils_acct.print_table('select * from scott.dept')
DEPTNO                         : 10
DNAME                          : ACCOUNTING
LOC                            : NEW YORK


PL/SQL procedure successfully completed.
```

Это показывает, что пользователь **SCOTT** может использовать процедуру, и она имеет доступ к объектам в схеме пользователя **SCOTT**. Однако при выполнении от имени пользователя **ANOTHER_USER** обнаруживается следующее:

```
scott@TKYTE816> connect another_user/another_user

another_user@TKYTE816>desc scott.dept
ERROR:
ORA-04043: object scott.dept does not exist

another_user@TKYTE816> set serverout on
another_user@TKYTE816>exec utils_acct.print_table('select * from
-> scott.dept');
BEGIN utils_acct.print_table('select * fromscott.dept'); END;
*
ERROR at line 1:
ORA-00942: table or view does not exist
```

```
ORA-06512: at "UTILS_ACCT.PRINT_TABLE", line 31
ORA-06512: at line 1
```

Пользователь, не имеющий доступа к таблицам пользователя **SCOTT**, не сможет использовать процедуру для получения доступа к ним. Для полноты эксперимента снова зарегистрируемся как **SCOTT** и предоставим пользователю **ANOTHER_USER** соответствующую привилегию:

```
another_user@TKYTE816> connect scott/tiger

scott@TKYTE816>grant select on dept to another_user;

Grant succeeded.

scott@TKYTE816> connect another_user/another_user

another_user@TKYTE816>executils_acct.print_table('select * from
scott.dept');
DEPTNO                          : 10
DNAME                           : ACCOUNTING
LOC                             : NEW YORK


PL/SQL procedure successfully completed.
```

Это демонстрирует практическое использование прав вызывающего в универсальных приложениях.

Приложения, работающие со словарем данных

Разработчикам всегда хотелось создать процедуры, выдающие информацию из словаря данных в более удобном виде, чем можно получить с помощью простого оператора **SELECT,** или, например, средства получения операторов ЯОД. С помощью процедур, работающих с правами создателя, сделать это было очень сложно. Если используются представления **USER_*** (например, **USER_TABLES),** будет выдаваться информация об объектах, принадлежащих создателю процедуры, а не вызывающему. Дело в том, что в условиях всех представлений USER_* и **ALL_*** есть конструкция:

```
where o.owner# = userenv('SCHEMAID')
```

Функция **USERENV('SCHEMAID')** возвращает идентификатор пользователя для схемы, в которой выполняется процедура. В хранимой процедуре с правами создателя (т.е. в стандартной хранимой процедуре) это значение постоянно — это всегда будет идентификатор пользователя, в схеме которого создана процедура. Это означает, что если кто-то напишет процедуру, обращающуюся к словарю данных, эта процедура будет видеть **его** объекты, а не объекты пользователя, выполняющего запрос. Более того, в хранимой процедуре **роли не используются** (мы рассмотрим эту проблему чуть позже), так что, если доступ к таблице в схеме другого пользователя получен через роль, в хранимой процедуре эта таблица будет недоступна. Когда-то единственным решением было создание хранимой процедуры, обращающейся к представлениям **DBA_*** (после получения **непосредственных** привилегий для этого) и реализующей собственный механизм защиты, который гарантировал получение пользователями только той информации,

которая доступна им в представлениях **ALL_*** или **USER_*.** Это более чем нежелательно, поскольку приходится писать большой объем кода, предоставлять права доступа ко всем представлениям **DBA_*;** кроме того, при малейшей невнимательности процедура позволит получить несанкционированный доступ к объектам.

Здесь поможет процедура, работающая с правами вызывающего. Теперь можно не только создать хранимую процедуру, обращающуюся к представлениям **ALL_*** и **USER_***, — это можно делать от имени текущего зарегистрированного пользователя, с его привилегиями и даже ролями. Мы продемонстрируем это, реализовав "усовершенствованную" команду **DESCRIBE.** Это будет минимальная по возможностям реализация — разобравшись, как это работает, вы сможете добавить любые возможности:

```
tkyte@TKYTE816> create or replace
  2  procedure desc_table(p_tname in varchar2)
  3  AUTHID CURRENT_USER
  4  as
  5  begin
  6    dbms_output.put_line('Типа    данных для таблицы ' || p_tname);
  7       dbms_output.new_line;
  8
  9      dbms_output.put_line(rpad('Имя столбца',31)  ||
 10               rpad('Тип               данных',20)     ||
 11                rpad('Длина',11)                 ||
 12                             'Пустые значения');
 13   dbms_output.put_line(rpad('-\30,'-')    ||  '  '  ||
 14              rpad('-',19,'-')                ||  '  '  ||
 15              rpad('-',10,'-')                ||  '  '  ||
 16                 '————————————————' ) ;
 17     for x in
 18      (select column_name,
 19               data_type,
 20               substr(
 21               decode(data_type,
 22                      'NUMBER', decode(data_precision, NULL, NULL,
 23                           '('||data_precision||','||data_scale||')'),
 24                       data_length),1,11) data_length,
 25     decode(nullable,'Y','null','not      null')      nullable
 26          from user_tab_columns
 27         where table_name = upper(p_tname)
 28         order by column_id)
 29      loop
 30          dbms_output.put_line(rpad(x.column_natne,31) ||
 31                               rpad(x.data_type,20)     ||
 32                                rpad(x.data_length,11) ||
 33                                x.nullable);
 34      end loop;
 35
 36     dbms_output.put_line(chr(10)  || chr(10)  ||
 37                          'Индексы по ' || p_tname);
 38
 39     for z in
 40      (select a.index_name, a.uniqueness
```

```
          from user_indexes a
         where a.table_name - upper(p_tname)
           and index_type - 'NORMAL')
      loop
          dbms_output.put(rpad(z.index_name,31) ||
                                      z.uniqueness);
         for y in
     (selectdecode(column_position,l,'(',' ' ) | |
                                              column_name column_name
              from user_ind_columns b
             where b.index_name = z.index_name
             order by column_position)
          loop
              dbms_output.put(y.column_name);
          end loop;
        dbms_output.put_line(1)' II chr(10));
      end loop;

end;
/
Procedure created.

tkyte@TKYTE816> grant execute on desc_table to public
  2  /

Grant succeeded.
```

Эта процедура интенсивно обращается к представлениям **USER_INDEXES** и **USER_IND_COLUMNS.** При работе с правами создателя (без конструкции **AUTHID CURRENT_USER)** эта процедура сможет выдать информацию только для **одного** пользователя (и всегда — для одного и того же). Однако при использовании прав вызывающего процедура будет выполняться от имени и с привилегиями пользователя, зарегистрировавшегося во время выполнения. Так что, хотя процедура и принадлежит пользователю **TKYTE,** ее можно выполнять от имени пользователя **SCOTT** и получать результат, подобный следующему:

```
tkyte@TKYTE816> connect scott/tiger

scott@TKYTE816> set serveroutput on format wrapped
scott@TKYTE816> exec tkyte.desc_table('emp')
Типы данных для таблицы emp

Имя столбца            Тип данных         Длина      Пустые значения

EMPNO                  NUMBER              (4,0)      not null
ENAME                  VARCHAR2           10          null
JOB                    VARCHAR2           9           null
MGR                    NUMBER              (4,0)      null
HIREDATE               DATE               7           null
SAL                    NUMBER              (7,2)      null
COMM                   NUMBER              (7,2)      null
DEPTNO                 NUMBER              (2,0)      null
```

```
Индексы по emp
EMP_PK                  UNIQUE(EMPNO)

PL/SQL procedure successfully completed.
```

Универсальные объектные типы

Идея в данном случае та же, что и в предыдущем, но результаты могут быть более общезначимыми. С помощью средств Oracle 8, позволяющих создавать собственные объектные типы со специфическими методами обработки данных, можно создавать методы, работающие с набором привилегий текущего зарегистрированного пользователя. То есть, создав универсальные, обобщенные типы, их устанавливают в базе данных один раз и разрешают всем использовать. Без возможности работать с правами вызывающего владелец объектного типа должен был иметь очень мощные привилегии (как было описано ранее) или надо было устанавливать этот объектный тип в каждой схеме, где его предполагалось использовать.

Именно с правами вызывающего всегда работали объектные типы, стандартно поставляемые в составе сервера Oracle (например, типы **ORDSYS**.*, используемые для поддержки компонентов interMedia), что позволяло устанавливать их в базе данных только один раз, а использовать — любому пользователю со своими привилегиями. Это имеет значение, потому что объектные типы **ORDSYS** выполняют чтение и запись в таблицы базы данных. Набор таблиц, к которым они обращаются, полностью зависит от того, кто именно выполняет соответствующие методы. Именно это свойство позволяет обеспечить универсальность и общедоступность этих типов. Они устанавливаются в схеме **ORDSYS**, но пользователь **ORDSYS** не имеет доступа к таблицам, с которыми они фактически работают. Теперь, используя Oracle 8i, разработчики приложений могут создавать такие же типы.

Реализация собственных средств контроля доступа

В Oracle 8i появилась возможность тщательного контроля доступа (**F**ine **G**rained **A**ccess **C**ontrol — **FGAC**), благодаря чему можно реализовать правила защиты, предотвращающие несанкционированный доступ к данным. Обычно для этого в каждую таблицу добавляется столбец, например **COMPANY**. Значения в этом столбце автоматически формируются триггером, а в каждый запрос включается условие **WHERE COMPANY = SYS_CONTEXT (...)**, ограничивающее набор доступных пользователю строк только теми, доступ к которым авторизован (подробнее об этом см. в главе 21).

Можно также создавать отдельную схему (набор таблиц) для каждой компании. Другими словами, для каждой компании устанавливается и наполняется данными свой набор таблиц. При этом в принципе невозможен доступ одного пользователя к данным другого, поскольку данные эти физически хранятся в другой таблице. Это — вполне уместный подход, имеющий преимущества (и недостатки) по сравнению со средствами тщательного контроля доступа. Проблема, однако, в том, что хотелось бы поддерживать один набор хранимого кода для всех пользователей. Кэширования же в разделяемом пуле

десяток копий одного и того же большого PL/SQL-пакета желательно избежать. Не хотелось бы изменять десяток экземпляров одного и того же кода, если в нем будет найдена ошибка. Не хотелось бы, чтобы пользователи выполняли потенциально разные версии одного кода. Работа с правами вызывающего идеально поддерживает эту модель защиты (несколько наборов таблиц и один экземпляр кода).

Имея возможность работать с правами вызывающего, можно написать одну хранимую процедуру, обращающуюся к таблицам с правами доступа текущего пользователя и разрешением имен в его схеме. Как было продемонстрировано в примере с процедурой **PRINT_TABLE**, это можно сделать как с помощью динамического, так и статического SQL Рассмотрим следующий пример. Установим таблицы **EMP/DEPT** в схеме **SCOTT** и в моей схеме **TKYTE**. Третий пользователь будет создавать приложение, использующее таблицы **EMP** и **DEPT** для создания отчета; он не будет иметь доступа к таблицам **EMP** и **DEPT** ни в схеме **SCOTT**, ни в схеме **TKYTE** (его таблицы созданы для тестирования). Вы увидите, что процедура, выполненная пользователем **SCOTT**, будет выдавать данные из схемы **SCOTT**; когда же ее выполнит пользователь **TKYTE**, будут использованы таблицы последнего:

```
tkyte@TKYTE816> connect scott/tiger
scott@TKYTE816> grant select on emp to public;
Grant succeeded.
scott@TKYTE816> grant select on dept to public;
Grant succeeded.
scott@TKYTE816> connect tkyte/tkyte
tkyte@TKYTE816> create table dept as select * from scott.dept;
Table created.
tkyte@TKYTE816> create table emp as select * from scott.emp;
Table created.
tkyte@TKYTE816> insert into emp select * from emp;
14 rows created.
tkyte@TKYTE816> create user application identified by pw
  2          default tablespace users quota unlimited on users;
User created.
tkyte@TKYTE816> grant create session, create table,
  2                create procedure to application;
Grant succeeded.
tkyte@TKYTE816> connect application/pw
application@TKYTE816> create table emp as select * from scott.emp where 1=0;
Table created.
application@TKYTE816> create table dept as
  2                  select * from scott.dept where 1=0;
Table created.
```

Итак, имеются три пользователя, у каждого из которых собственная пара таблиц EMP/DEPT. Данные же в этих таблицах существенно отличаются. У пользователя SCOTT — обычный набор данных EMP, у пользователя TKYTE данных в два раза больше, а у пользователя APPLICATION эти таблицы пусты. Теперь создадим приложение:

```
application@TKYTE816> create or replace procedure emp_dept_rpt
  2  AUTHID CURRENT_USER
  3  as
  4  begin
  5  dbms_output.put_line('Зарплаты  и количество сотрудников по
-> отделам') ;
  6    dbms_output.put_line(chr(9)||'Отдел    Зарплата    Количество');
  7     dbms_output.put_line(chr(9)          ||          '----------') ;
  8     for x in (select dept.deptno, sum(sal) sal, count(*) cnt
  9               from emp, dept
 10             where dept.deptno = emp.deptno
 11             group by dept.deptno)
 12    loop
 13       dbms_output.put_line(chr(9)  ||
 14            to_char(x.deptno,'99999') || ' ' ||
 15            to_char(x.sal,'99,999') || ' ' ||
 16             to_char(x.cnt,'99,999'));
 17    end loop;
 18  dbms_output.put_line('==================') ;
 19 end;
 20 /

Procedure created.

application@TKYTE816> grant execute on emp_dept_rpt to public
  2  /

Grant succeeded.

application@TKYTE816> set serveroutput on format wrapped
application@TKYTE816> exec emp_dept_rpt;
Зарплаты и количество сотрудников по отделам
        Отдел   Зарплата   Количество



PL/SQL procedure successfully completed.
```

Когда процедуру выполняет пользователь APPLICATION таблицы пусты, как и ожидалось. При выполнении же этого приложения пользователями SCOTT и TKYTE:

```
tkyte@TKYTE816> connect scott/tiger

scott@TKYTE816> set serveroutput on format wrapped
scott@TKYTE816> exec application.ernp_dept_rpt
Зарплаты и количество сотрудников по отделам
        Отдел   Зарплата   Количество

            10    8,750        3
            20   10,875        5
            30    9,400        6
```

```
PL/SQL procedure successfully completed.

scott@TKYTE816>connect tkyte/tkyte
tkyte@TKYTE816>set serveroutput on format wrapped
tkyte@TKYTE816>exec  application.emp_dept_rpt
Зарплаты и количество сотрудников по отделам
       Отдел   Зарплата   Количество

          10  17,500       6
          20  21,750      10
          30  18,800      12

PL/SQL procedure successfully completed.
```

Как видите, процедура действительно обращается к разным таблицам в разных схемах. Тем не менее, как будет показано в разделе "Проблемы", надо позаботиться о синхронизации этих схем. Не только должны существовать таблицы с соответствующими именами, но типы данных, порядок и количество столбцов в них при использовании статического SQL тоже должны совпадать.

Когда использовать права создателя

Работа с правами создателя продолжает оставаться доминирующим методом использования скомпилированных хранимых объектов. Для этого имеются две основные причины.

- **Производительность.** База данных, в которой используются процедуры с правами создателя, существенно лучше масштабируется и обеспечивает более высокую производительность, чем база данных, использующая процедуры с правами вызывающего.
- **Защита.** Процедуры с правами создателя имеют такие особенности с точки зрения защиты, которые делают их применение единственно верным выбором почти во всех случаях.

Производительность и масштабируемость

Процедура, работающая с правами создателя, — замечательная вещь с точки зрения защиты и производительности. В разделе "Как работают процедуры с правами вызывающего" будет показано, что, благодаря статическому связыванию на этапе компиляции, можно существенно повысить эффективность во время выполнения. Все проверки защиты, зависимостей и т.п. выполняются один раз, при компиляции. Для процедуры, работающей с правами вызывающего, большая часть этих действий будет делаться во время выполнения. Более того, может потребоваться многократно выполнять эти действия в одном сеансе, после выполнения оператора **ALTER SESSION** или **SET ROLE.** Любое изменение среды выполнения приводит к изменению поведения процедуры с правами вызывающего. Процедура же с правами создателя от этих изменений не зависит.

Кроме того, как будет показано далее в разделе "Проблемы", процедура, работающая с правами вызывающего, более интенсивно использует разделяемый пул, чем ана-

логичная процедура, работающая с правами создателя. Поскольку среда выполнения для процедуры с правами создателя статична, **все** выполняемые ею статические SQL-операторы могут совместно использоваться в разделяемом пуле. Как было показано в других главах этой книги, необходимо заботиться о правильном использовании разделяемого пула (использовать связываемые переменные, избегать излишних разборов и т.д.). Использование процедур с правами создателя гарантирует максимально эффективное использование разделяемого пула. Процедуры же с правами вызывающего могут приводить к неэффективному использованию разделяемого пула. Вместо того чтобы один запрос, **SELECT * FROM T,** при использовании в процедуре означал бы одно и то же для всех пользователей, он может иметь для пользователей разный смысл. В разделяемом пуле будет больше различных SQL-операторов. Использование прав создателя обеспечивает более эффективное использование разделяемого пула.

Защита

Используя права создателя, можно создать процедуру, безопасно и корректно обрабатывающую определенный набор объектов базы данных. Затем можно предоставить другим пользователям возможность выполнять эту процедуру с помощью оператора **GRANT EXECUTE ON <процедура> TO <пользователь>/public/<роль>.** Эти пользователи смогут запускать процедуру для чтения/записи таблиц (способом, предусмотренным в коде этой процедуры), но никаким другим способом читать или записывать данные в наши таблицы они не могут. Другими словами, мы только что создали надежный процесс изменения или чтения объектов безопасным образом и теперь можем предоставлять пользователям право на это, не опасаясь, что они смогут каким-либо другим способом прочитать или изменить эти объекты. Они не смогут вставлять записи в таблицу сотрудников с помощью утилиты SQL*Plus. Это можно будет делать **только** с помощью хранимой процедуры, реализующей все необходимые проверки. Этот способ работы существенно влияет на разработку приложения и предоставление прав на использование ваших данных. Больше не придется выполнять операторы **GRANT INSERT** для таблицы, как это делалось для клиент-серверного приложения, непосредственно выполняющего SQL-операторы **INSERT.** Вместо этого придется выполнить оператор **GRANT EXECUTE** для процедуры, которая может проверять и оценивать данные и их защищенность. Беспокоиться о целостности данных при этом тоже больше не нужно (процедура точно задает, что и как необходимо делать, а других способов работы с данными просто нет).

Сравните это с работой типичных клиент-серверных и даже многих 3-уровневых приложений. В клиент-серверном приложении операторы **INSERT, UPDATE, DELETE** и т.п. включены непосредственно в код клиентского приложения. Для работы этого приложения, пользователю необходимо предоставить привилегии **INSERT, UPDATE** и **DELETE** непосредственно для базовых таблиц. Теперь весь мир имеет доступ к базовым таблицам через любой интерфейс, способный взаимодействовать с СУБД Oracle. При использовании процедуры с правами создателя такой проблемы нет. **Единственный** способ изменения таблиц — с помощью надежной процедуры, которой вполне можно доверять. Это очень важное свойство.

Часто разработчики спрашивают: "Как сделать так, чтобы только мое приложение, **myapp.exe,** могло выполнять действие X в базе данных?". Например, надо, чтобы это приложение могло выполнять вставку в таблицу, но другие приложения этого сделать не могли. **Единственно безопасный способ** сделать это — поместить алгоритмы работы с данными приложения **myapp.exe** в базу данных, и никаких операторов **INSERT**, **UPDATE**, **DELETE** или **SELECT** в клиентском приложении. Разместив приложение непосредственно в базе данных, устранив необходимость выполнять вставку (или любые другие операции с таблицей) в клиентском приложении, вы добьетесь того, чтобы **только** одно приложение могло обращаться к данным. Размещая алгоритмы работы с базой данных приложения в ней самой, мы делаем из приложения просто еще одни уровень абстракции. Не имеет значения, как вызывается приложение (его компонент, работающий с базой данных) — из SQL*Plus, из графического интерфейса или из другого, еще нереализованного интерфейса, — в базе данных работает одно и то же приложение.

Как работают процедуры с правами вызывающего

Именно тут возможно непонимание: когда и какие привилегии действуют. Прежде чем приступить к рассмотрению функционирования процедур с правами вызывающего, рассмотрим, как работают процедуры **с правами создателя.** Разобравшись в этом, мы рассмотрим, чем отличается работа процедур с правами вызывающего при различных условиях вызова.

Права создателя

При использовании прав создателя хранимая процедура компилируется с учетом привилегий, непосредственно предоставленных пользователю, "владеющему" процедурой. Под "непосредственно предоставленными привилегиями" подразумеваются все объектные и системные привилегии, предоставленные пользователю или роли **PUBLIC,** но не другим ролям, которые предоставлены пользователю или роли **PUBLIC.** Короче, для процедур с правами создателя роли не учитываются и не используются, ни во время компиляции, ни при выполнении. Процедура компилируется только с учетом непосредственных привилегий. Это описано в руководстве *OracleApplication Developer's Guide mak:*.

Привилегии, необходимые для создания процедур и функций

При создании отдельной процедуры или функции, спецификации или тела пакета должны выполняться следующие требования.

Необходимо наличие системной привилегии **CREATE PROCEDURE** (для создания процедуры или пакета в своей схеме) или системной привилегии **CREATE ANY PROCEDURE** (для создания процедуры или пакета в другой пользовательской схеме).

Внимание: Для успешной компиляции процедуры или пакета требуются следующие дополнительные привилегии:

- владелец процедуры или пакета должен явно получить необходимые объектные привилегии для всех объектов, на которые есть ссылки в коде;

- владелец не может получить необходимые привилегии через роли.

Если привилегии владельца процедуры или пакета изменяются, процедуру необходимо повторно аутентифицировать перед выполнением. Если необходимая для доступа к объекту привилегия у владельца процедуры (или пакета) отобрана, выполнение процедуры становится невозможным.

Хотя это явно и не сказано, предоставление привилегии роли **PUBLIC** ничуть не хуже, чем непосредственно владельцу процедуры. Необходимость непосредственного предоставления привилегий владельцу процедуры, работающей с правами создателя, иногда приводит к странным ситуациям. Оказывается, можно выполнить запрос к объекту в SQL*Plus и использовать анонимный блок для доступа к этому же объекту, но нельзя создать хранимую процедуру для обращения к этому объекту. Зададим необходимые для данного примера привилегии:

```
scott@TKYTE816> revoke select on emp from public-
Revoke succeeded.
scott@TKYTE816> grant select on emp to connect;
Grant succeeded.
scott@TKYTE816> connect tkyte/tkyte
tkyte@TKYTE816> grant create procedure to another_user;
Grant succeeded.
```

А теперь убедимся, что пользователь ANOTHER_USER может делать запросы к таблице SCOTT.EMP:

```
tkyte@TKYTE816> connect another_user/another_user

another_user@TKYTE816> select count(*) from scott.emp;
  COUNT(*)

        14
```

Пользователь **ANOTHER_USER** также может выполнять анонимный блок PL/SQL:

```
another_user@TKYTE816> begin
  2    for x in (select count(*) cnt from scott.emp)
  3    loop
  4          dbms_output.put_line(x.cnt);
  5    end loop;
  6  end;
  7  /
14
PL/SQL procedure successfully completed.
```

Однако при попытке создать процедуру, идентичную представленному выше PL/SQL-блоку, мы получим следующее:

```
another_user@TKYTE816> create or replace procedure P
  2  as
```

```
  3 begin
  4   for x in (select count(*) cnt from scott.emp)
  5    loop
  6          dbms_output.put_line(x.cnt);
  7    end loop;
  8 end;
  9 /

Warning: Procedure created with compilation errors.

another_user@TKYTE816> show err
Errors for PROCEDURE P:

LINE/COL ERROR

4/14    PL/SQL: SQL Statement ignored
4/39    PLS-00201: identifier 'SCOTT.EMP' must be declared
6/9     PL/SQL: Statement ignored
6/31    PLS-00364: loop index variable 'X' use is invalid
```

Я не могу создать процедуру (собственно, любой скомпилированный хранимый объект, скажем, представление или триггер), обращающуюся к таблице **SCOTT.EMP**. Это предсказуемая и описанная в документации особенность. В рассмотренном выше примере пользователь **ANOTHER_USER** имеет роль **CONNECT**. Роли **CONNECT** была предоставлена привилегия **SELECT** для таблицы SCOTT.EMP. Эта привилегия роли **CONNECT**, однако, недоступна в хранимой процедуре с правами создателя, поэтому и выдается сообщение об ошибке. Чтобы избежать таких странностей, я рекомендую выполнить оператор **SET ROLE NONE** в среде SQL*Plus и попытаться выполнить оператор, который предполагается включить в хранимую процедуру. Например:

```
another_user@TKYTE816> set role none;

Role set.

another_user@TKYTE816> select count(*) from scott.emp;
select count(*) from scott.emp
                      *

ERROR at line 1:
ORA-00942: table or view does not exist
```

Если оператор сработает в SQL*Plus без ролей, он, несомненно, сработает и в хранимой процедуре, выполняющейся с правами создателя.

Компиляция процедуры с правами создателя

При компиляции процедуры выполняется несколько действий, связанных с привилегиями. Здесь я их вкратце опишу, а затем рассмотрю подробно.

- Проверяется существование всех объектов, к которым процедура обращается статически (всех, к которым не обращаются с помощью динамического SQL). Имена разрешаются с помощью стандартных правил области действия по отношению к владельцу процедуры.
- Проверяется, все ли объекты доступны в нужном режиме. Например, если выполняется оператор **UPDATE T**, сервер Oracle проверит, может ли создатель или

роль **PUBLIC** выполнять **UPDATE** T непосредственно, без использования каких-либо ролей.

- Устанавливается и поддерживается зависимость процедуры от объектов, на которые она ссылается. Если процедура выполняет оператор **SELECT FROM** T, регистрируется зависимость процедуры от таблицы T.

Если, например, создается процедура P, пытающаяся выполнить оператор SELECT * **FROM T,** компилятор сначала преобразует T в полностью уточненное имя. Имя T в базе данных неоднозначно — таких таблиц представлений может быть несколько. Чтобы выяснить, какую именно таблицу T использовать, сервер Oracle применяет правила определения области действия. Вместо синонимов подставляются соответствующие базовые объекты, причем для каждого объекта указывается имя соответствующей схемы. Это разрешение имен выполняется для текущего зарегистрированного пользователя (создателя). Другими словами, сервер ищет объект T у данного пользователя и использует его (при этом используются приватные синонимы пользователя), затем сервер ищет T среди общедоступных синонимов и т.д.

Определив объект, на который ссылается имя T, сервер Oracle определяет, возможен ли доступ к этому объекту в нужном режиме. В данном случае, если объект T принадлежит создателю процедуры или создатель непосредственно получил привилегию **SELECT** на объект T (или эта привилегия была предоставлена роли **PUBLIC),** процедура будет скомпилирована. Если создатель процедуры не имеет доступа к объекту по имени T благодаря непосредственно предоставленной привилегии, процедура P не будет скомпилирована. Таким образом, когда объект (хранимая процедура, ссылающаяся на T) компилируется, сервер Oracle выполняет все эти проверки. Если они "пройдены", сервер Oracle компилирует процедуру, сохраняет двоичный код процедуры и устанавливает зависимости между этой процедурой и объектом T. Эта зависимость используется для проверки действительности процедуры в дальнейшем, если что-то произошедшее с объектом T может потребовать перекомпиляции процедуры. Например, если позже мы выполним **REVOKE SELECT ON T** и отберем эту привилегию у владельца процедуры, сервер Oracle пометит все хранимые процедуры этого пользователя, зависящие от объекта T и ссылающиеся на T, как недействительные **(INVALID).** Если мы добавим столбец с помощью оператора **ALTER T ADD** ..., сервер Oracle сделает недействительными все зависящие от него процедуры. Это приведет к их автоматической перекомпиляции при следующем вызове.

Интересно разобраться не только в том, что сохраняется, но и что не сохраняется при компиляции объекта. Сервер Oracle не сохраняет информацию о привилегии, необходимой для получения доступа к объекту T. Мы знаем только, что процедура P зависит от T. Мы не знаем, почему получен доступ к объекту T:

- потому что создателю процедуры была предоставлена соответствующая привилегия (**GRANT SELECT ON T TO USER**);
- потому что привилегия была предоставлена роли **PUBLIC** (**GRANT SELECT ON T TO PUBLIC);**
- потому что пользователь имеет привилегию SELECT ANY TABLE.

Знать, что не сохраняется при компиляции, интересно потому, что, если одна из этих привилегий будет отобрана, процедура P станет недействительной. Если при компиляции процедуры у пользователя были все три привилегии, лишение пользователя **любой** из них приведет к пометке процедуры как недействительной и принудительной перекомпиляции перед следующим выполнением.

Теперь, когда процедура скомпилирована и хранится в базе данных, а все зависимости учтены, можно выполнять процедуру, точно зная, что представляет собой объект T, и будучи уверенными, что он доступен. Если что-то способное повлиять на доступность объекта T произойдет с самим объектом T или с набором базовых привилегий создателя процедуры, процедура станет недействительной и ее придется перекомпилировать.

Права создателя и роли

Теперь нам предстоит разобраться, почему при компиляции и выполнении хранимых процедур с правами создателя роли не учитываются. Сервер Oracle не хранит информацию о том, почему мы можем обращаться к объекту T, — только то, что **мы это можем.** Любое изменение привилегий, которое может сделать невозможным обращение к объекту T, приведет к пометке процедуры как недействительной и ее перекомпиляции. Если роли не учитываются, такое изменение привилегий может произойти только при выполнении операторов **REVOKE SELECT ANY TABLE** или **REVOKE SELECT ON T** для пользователя-создателя или роли **PUBLIC.**

Если роли учитываются, набор операторов, которые могут сделать процедуру недействительной, существенно расширяется. Представьте на минуту, что роли позволяют обращаться к объектам из хранимых процедур. Тогда при изменении **любой** из ролей, лишении ее привилегии или роли (ведь роли можно предоставлять ролям), мы рискуем сделать недействительными множество процедур (даже тех, которые не использовали привилегии измененной роли).

Рассмотрим результат лишения роли системной привилегии. Это будет аналогично лишению роли **PUBLIC** мощной системной привилегии (не делайте этого, просто представьте, а если уж хотите сделать, то в тестовой базе данных). Если роли **PUBLIC** была предоставлена привилегия **SELECT ANY TABLE,** в результате лишения ее этой привилегии будут помечены как недействительные практически все процедуры в базе данных. Если процедуры зависят от ролей, любая процедура в базе данных затрагивается даже наименьшими изменениями прав доступа. Поскольку одно из основных преимуществ хранимых процедур — однократная компиляция при многократном выполнении, это крайне негативно повлияет на производительность.

Учтите также следующие особенности ролей.

- **Роли могут быть нестандартными.** Если я создам нестандартную роль, включу ее и скомпилирую процедуру, работа которой зависит от привилегий этой роли, по завершении сеанса у меня этой роли больше не будет. Станет ли при этом процедура **недействительной?** Почему? А почему — нет? Я легко могу обосновать оба варианта.
- **Роли могут быть защищены паролями.** Если изменен пароль роли, надо ли перекомпилировать все объекты, которым эта роль может понадобиться? Мне эта роль может быть предоставлена, но, не зная ее нового пароля, я не смогу ею восполь-

зоваться. Будут ли по-прежнему доступны соответствующие привилегии? Почему — да или почему — нет? Есть аргументы и за, и против.

Подведем итоги по использованию ролей в процедурах, работающих с правами создателя.

- Вы можете работать с тысячами или десятками тысяч пользователей. Они не создают хранимые объекты. Для управления всеми этими пользователям необходимы роли. Именно для этого и создавались роли.
- Вы можете использовать намного меньше схем приложений (в них и находятся хранимые объекты). Нужно точно знать, какие для них необходимы привилегии и почему. С точки зрения защиты это называется концепцией *минимальных привилегий.* Надо явно указать, какие привилегии нужны и зачем. Если унаследовано множество привилегий от ролей, добиться минимальности привилегий практически невозможно. При явном задании привилегий не возникают проблемы, поскольку количество схем приложений **невелико** (но количество их пользователей **огромно).**
- Наличие непосредственной взаимосвязи между создателем и процедурой позволяет сделать базу данных намного эффективней. Мы перекомпилируем объекты только в случае **необходимости.** Это существенно повышает эффективность их работы.

Права вызывающего

Между процедурами с правами вызывающего и процедурами с правами создателя (и анонимными блоками PL/SQL) есть существенное отличие с точки зрения использования привилегий и разрешения ссылок на объекты. Что касается выполнения SQL-операторов, процедуры с правами вызывающего подобны анонимному блоку PL/SQL, но PL/SQL-операторы выполняются в них так же, как в процедурах с правами вызывающего. Кроме того, роли **могут** учитываться в процедуре с правами вызывающего, в зависимости от того, как к ней обращаются (в отличие от процедуры с правами создателя, которая игнорирует роли при доступе к объектам).

Рассмотрим две части процедур, работающих с правами вызывающего:

- "SQL-части" - все операторы **SELECT**, **INSERT**, **UPDATE**, **DELETE** и все операторы, динамически выполняемые с помощью **DBMS_SQL** или **EXECUTE IMMEDIATE** (включая динамически выполняемый PL/SQL-код);
- "PL/SQL-части" — статические ссылки на объектные типы в объявлениях переменных, вызовы хранимых процедур, пакетов, функций и т.п.

В процедурах с правами вызывающего обработка этих "частей" очень отличается. Имена в "SQL-части" разрешаются при компиляции (для определения структур данных и т.п.) и еще раз — при выполнении. Именно это позволяет хранимой процедуре с запросом **SELECT * FROM EMP** обращаться к другим таблицам **EMP** при выполнении другими пользователями. Однако "PL/SQL-части" при компиляции связываются статически, как

и в процедуре с правами создателя. Поэтому, если в процедуре с правами вызывающего имеется следующий код:

```
AUTHID CURRENT_USER
as
begin
   for x in (select * from T) loop
         proc(x.cl);
   end loop;
```

то ссылки на T будут разрешаться во время выполнения (как и во время компиляции, чтобы понять, что означает SELECT *) динамически, что позволяет использовать разные объекты T для каждого пользователя. Ссылка на процедуру PROC, однако, будет разрешена только при компиляции, поэтому процедура будет статически связана с одной процедурой PROC. Пользователю, вызывающему эту процедуру, не нужна привилегия EXECUTE ON PROC, но вот привилегия SELECT для объекта T нужна. Не хочется вас запутывать, но если необходимо разрешать вызов PROC при выполнении, есть механизм и для этого. Можно написать следующий код:

```
AUTHID CORRENT_USER
as
begin
   for x in (select * from T) loop
         execute immediate 'begin proc(:x); end;' USING x.cl;
   end loop;
```

В этом случае ссылка на процедуру **PROC** будут разрешаться на основе набора привилегий вызывающего, и этим вызывающим (или соответствующим ролям, если действуют роли) необходимо предоставить привилегию EXECUTE.

Разрешение ссылок и передача привилегий

Давайте рассмотрим, как проверяются привилегии в процедуре с правами вызывающего. Для этого придется рассмотреть различные среды, или стеки вызовов, в которых может вызываться процедура:

- непосредственный вызов пользователем;
- вызов из процедуры с правами создателя;
- вызов из другой процедуры с правами вызывающего;
- вызов из SQL-оператора;
- вызов из представления, ссылающегося на процедуру с правами вызывающего;
- вызов из триггера.

Для одной и той же процедуры результат в каждой из перечисленных сред может отличаться. В каждом из этих случаев процедура с правами вызывающего может обращаться при выполнении к другим таблицам и объектам базы данных.

Начнем с изучения того, как связываются объекты и какие привилегии доступны в процедуре с правами вызывающего при выполнении в каждой из перечисленных сред. Случаи представления и триггера будем считать одинаковыми, поскольку там процедура может работать только с правами создателя. Кроме того, поскольку статические объекты PL/SQL всегда, во всех средах **разрешаются во время выполнения,** мы их рассматривать не будем. Они всегда разрешаются в схеме и с привилегиями создателя. Текущему зарегистрированному пользователю не нужен доступ к объекту PL/SQL, на который делается ссылка. В следующей таблице описано предполагаемое поведение в каждой из сред:

Среда	*SQL-объекты и динамически вызываемый PL/SQL*	*Роли действуют?*
Непосредственный вызов пользователем. Например: SQL> exec p;	Ссылки на эти объекты разрешаются в стандартной схеме и со стандартными привилегиями текущего пользователя. Неуточненные ссылки на объекты будут разрешаться в их схеме. Все объекты должны быть доступны текущему зарегистрированному пользователю. Если процедура выполняет SELECT из T, текущий пользователь также должен иметь привилегию SELECT для объекта T (полученную либо непосредственно, либо через роль).	Да. Все роли, включенные до выполнения процедуры, действуют и в процедуре. Они будут использоваться для разрешения или запрещения доступа ко всем SQL-объектам и динамически вызываемому PL/SQL-коду.
Вызов из процедуры с правами создателя (P1), где P2 - процедура с правами вызывающего. Например: `procedure p1 is begin p2; end;`	Этот вызов разрешается в схеме создателя, в схеме вызывающей процедуры. Неуточненные имена объектов будут разрешаться в этой схеме, схеме вызывающей процедуры, а не в схеме текущего зарегистрированного пользователя и не в схеме, где была создана процедура с правами вызывающего, В нашем примере, владелец P1 всегда будет "вызывающим" внутри процедуры P2.	Нет. Роли не учитываются, поскольку вызвана процедура с правами создателя. В момент входа в процедуру с правами создателя все роли отключаются и не учитываются до выхода из этой процедуры,
Вызов из другой процедуры с правами вызывающего.	Аналогично непосредственному вызову пользователем.	Да. Точно так же, как и при вызове пользователем,
Вызов из SQL-оператора.	Аналогично непосредственному вызову пользователем.	Да. Точно так же, как и при вызове пользователем.

Среда	*SQL-объекты и динамически вызываемый PL/SQL*	*Роли действуют?*
Вызов из представления или триггера, ссылающегося на процедуру с правами вызывающего.	Аналогично вызову из процедуры с правами создателя.	Нет. Точно так же, как и при вызове из процедуры с правами создателя.

Как видите, среда выполнения может существенно влиять на выполнение процедуры с правами вызывающего. Одна и та же хранимая процедура на PL/SQL при непосредственном вызове и при вызове из другой хранимой процедуры может обращаться к различным наборам объектов, даже при регистрации от имени одного и того же пользователя.

Чтобы продемонстрировать это, я создам процедуру, показывающую, какие роли активны во время выполнения и обращающуюся к таблице за данными, свидетельствующими о "владельце" таблицы. Мы сделаем это для каждого из представленных выше случаев, за исключением вызова процедуры с правами вызывающего из другой процедуры с правами вызывающего, поскольку это ничем не отличается от ее непосредственного вызова. Начнем с создания двух учетных записей, которые будут использоваться для демонстрации:

```
tkyte@TKYTE816> drop user a cascade;
User dropped.
tkyte@TKYTE816> drop user b cascade;
User dropped.
tkyte@TKYTE816> create user a identified by a default tablespace data
temporary tablespace temp;
User created.
tkyte@TKYTE816> grant connect, resource to a;
Grant succeeded.
tkyte@TKYTE816> create user b identified by b default tablespace data
temporary tablespace temp;
User created.
tkyte@TKYTE816> grant connect, resource to b;
Grant succeeded.
```

Итак, созданы два пользователя, **A** и **B**, и каждому из них предоставлены две роли, **CONNECT** и **RESOURCE**. Пользователь A создаст процедуру с правами вызывающего, а также процедуру с правами создателя и представление, из которых будет вызываться процедура с правами вызывающего. При каждом выполнении процедура будет выдавать количество действующих ролей, имя текущего пользователя (с набором привилегий какой схемы она работает), имя текущей схемы и, наконец, какая таблица используется запросом. Начнем с создания таблицы, определенно принадлежащей пользователю **A**:

```
tkyte@TKYTE816> connect a/a
a@TKYTE816> create table t (x varchar2(255));
Table created.
a@TKYTE816> insert into t values ('Таблица пользователя A');
1 row created.
```

Затем пользователь A создает функцию с правами вызывающего, процедуру с правами создателя и представление:

```
a@TKYTE816> create function Invoker_rights_function return varchar2
  2  AUTHID CURRENT_USER
  3  as
  4          l_data varchar2(4000);
  5  begin
  6     dbms_output.put_line('Я - функция с правами вызывающего,
-> принадлежащая A');
  7     select 'current_user=' ||
  8            sys_context('userenv', 'current_user') ||
  9           ' current_schema=' ||
 10            sys_context('userenv', 'current_schema') ||
 11           ' active roles=' || cnt ||
 12           ' data from T=' || t.x
 13         into l_data
 14            from (select count(*) cnt from session_roles), t;
 15
 16     return l_data;
 17  end;
 18  /
Function created.
a@TKYTE816> grant execute on Invoker_rights_function to public-
Grant succeeded.
a@TKYTE816> create procedure Definer_rights_procedure
  2  as
  3          l_data varchar2(4000);
  4  begin
  5   dbms_output.put_line('Я   - процедура с правами создателя,
-> принадлежащая A') ;
  6     select 'current_user=' ||
  7            sys_context('userenv', 'current_user') ||
  8           ' current_schema=' ||
  9            sys_context('userenv', 'current_schema') ||
 10           ' active roles=' || cnt ||
 11           ' data from T=' | | t.x
 12         into l_data
 13        from (select count(*) cnt from session_roles), t;
 14
 15     dbms_output.put_line(l_data);
 16     dbms_output.put_line
-> ('Теперь вызываем функцию с правами вызывающего...');
 17     dbms_output.put_line(Invoker_rights_function);
```

```
18  end;
19  /
Procedure created.

a@TKYTE816> grant execute on Definer_rights_procedure to public-
Grant succeeded.

a@TKYTE816> create view V
  2  as
  3  select invoker_rights_function from dual
  4  /
View created.

a@TKYTE816> grant select on v to public
  2  /
Grant succeeded.
```

Зарегистрируемся как пользователь B, создадим таблицу T с идентифицирующей пользователя строкой и выполним созданную ранее функцию:

```
a@TKYTE816> connect b/b
b@TKYTE816> create table t (x varchar2(255));
Table created.
b@TKYTE816> insert into t values ("Таблица пользователя B');
1 row created.
b@TKYTE816> exec dbms_output.put_line(a.Invoker_rights_function)
Я — функция с правами вызывающего, принадлежащая A
current_user=B current_scheroa=B active roles=3 data from T=Таблица
-> пользователя B
PL/SQL procedure successfully completed.
```

Итак, мы видим: когда пользователь B непосредственно вызывает функцию с правами вызывающего, принадлежащую пользователю A, во время ее выполнения используются привилегии пользователя **B (current_user=B).** Далее, поскольку **current_schema** — тоже пользователь B, запрос выбирает данные из таблицы B.T, а не из A.T. Это доказывает строка **data from T=Таблица пользователя B** в представленных выше результатах. Наконец, мы видим, что при выполнении запроса в сеансе активны три роли (третья роль — **PLUSTRACE,** необходимая для использования **AUTOTRACE;** в моей базе данных она предоставлена роли **PUBLIC).** Посмотрим, что произойдет при вызове через процедуру с правами создателя:

```
b@TKYTE816> exec a.Definer_rights_procedure
Я — процедура с правами создателя, принадлежащая A
current_user=A current_schema=A active roles=0 data from T=Таблица
-> пользователя A
Теперь вызываем функцию с правами вызывающего...
Я — функция с правами вызывающего, принадлежащая A
current_user=A current_schema=A active roles=0 data from T=Таблица
-> пользователя A
PL/SQL procedure successfully completed.
```

Вы видите, что процедура с правами создателя выполняется с привилегиями пользователя A, кроме ролей **(active roles=0).** Процедура с правами создателя жестко связана с таблицей A.T и не обращается к таблице B.T.

Важнее всего то, что происходит при вызове функции с правами вызывающего из процедуры с правами создателя. Обратите внимание, что на этот раз **вызывающий** - пользователь A, а не B. Вызывающий определяется текущей схемой в момент вызова процедуры с правами вызывающего. Функция больше не выполняется от имени B, как в предыдущем случае, — теперь она выполняется от имени пользователя A. Поскольку **current_user** и **current_schema** теперь задают пользователя A, функция с правами вызывающего обращается к таблице пользователя A. Еще один важный факт: на этот раз роли в функции с правами вызывающего не действуют. При входе в процедуру с правами создателя роли отключаются и остаются отключенными до выхода из этой процедуры.

Теперь рассмотрим последствия вызова функции с правами вызывающего из SQL-оператора:

```
b@TKYTE816> select a.invoker_rights_function from dual;

INVOKER_RIGHTS_FUNCTION

current_user=B current_schema=B active roles=3 data from T=Таблица
-> пользователя B

b@TKYTE816> select * from a.v;

INVOKER_RIGHTS_FUNCTION

current_user=A current_schema=A active roles=0 data from T=Таблица
-> пользователя A
```

Как видите, вызов процедуры с правами вызывающего непосредственно из SQL-оператора (как в нашем случае, когда мы выбирали значение функции из таблицы DUAL), ничем не отличается от непосредственного вызова. Более того, вызов функции из представления, как во втором запросе, показывает, что она ведет себя так же, как при вызове из процедуры с правами создателя, поскольку представления всегда сохраняются с правами создателя.

Компиляция процедуры с правами вызывающего

А сейчас разберемся, что происходит при компиляции и сохранении в базе данных процедуры с правами вызывающего. Может показаться странным, но происходит в точности то же, что и при компиляции процедуры с правами создателя. Выполняются следующие шаги:

- Проверяется существование всех объектов, к которым процедура обращается статически (всех, к которым она не обращается с помощью динамического SQL). Имена разрешаются с помощью стандартных правил области действия по отношению к владельцу процедуры. Роли не учитываются.
- Проверяется, все ли объекты доступны в нужном режиме. Например, если выполняется оператор **UPDATE T,** сервер Oracle проверит, может ли создатель или роль **PUBLIC** выполнять **UPDATE T** непосредственно, не используя ролей.

- Устанавливается и поддерживается зависимость процедуры от объектов, на которые она ссылается. Если процедура выполняет оператор **SELECT FROM T,** регистрируется зависимость процедуры от таблицы T.

Это означает, что подпрограмма с правами вызывающего **при компиляции** обрабатывается точно так же, как подпрограмма с правами создателя. Многих это сбивает с толку. Они слышали, что процедуры с правами вызывающего используют роли, и это так. Но (повторяю) они не используются в ходе компиляции. Это означает, что пользователь, компилирующий хранимую процедуру, ее владелец, должен иметь непосредственный доступ ко всем статически используемым таблицам. Вспомните пример из раздела "Права создателя", где было показано, что можно успешно выполнить **SELECT COUNT(*) FROM EMP** в SQL и в анонимном блоке PL/SQL, но такой же оператор в хранимой процедуре приводит к ошибке компиляции. То же самое произойдет и в подпрограмме с правами вызывающего. Правила, сформулированные в разделе *Privileges Required to Create Procedures and Functions* руководства *Oracle 8i Application Developer's Guide* остаются в силе: все равно необходим непосредственный доступ к базовым объектам.

Причина — в механизме зависимостей, используемом сервером Oracle. Если действие, выполняемое в базе данных (например, оператор **REVOKE),** делает процедуру с правами создателя недействительной, аналогичная процедура с правами вызывающего тоже становится недействительной. Различие между процедурами с правами вызывающего и создателя наблюдается только при выполнении. С точки зрения зависимостей, пометки процедур как недействительных и привилегий, необходимых владельцу процедуры, никаких различий нет.

Эту проблему можно обойти, и для большинства процедур с правами вызывающего проблема эта вообще не актуальна. Однако из-за нее иногда приходится создавать *объекты-шаблоны.* В следующем разделе мы рассмотрим, что такое объекты-шаблоны и как их использовать, чтобы избежать необходимости предоставления непосредственных привилегий.

Использование объектов-шаблонов

Теперь, зная, что при компиляции процедура с правами вызывающего не отличается от процедуры с правами создателя, можно понять, почему необходим непосредственный доступ ко всем объектам. При разработке процедур с правами вызывающего, в которых предполагается использование ролей, создателю необходимы непосредственные привилегии, а не роли. Их получение может оказаться невозможным по любой причине Достаточно, чтобы кто-то решил: "Привилегии select на эту таблицу я не дам"), и придется искать решение.

Тут пригодятся объекты-шаблоны. *Объект-шаблон* — это объект, к которому пользователь-создатель имеет непосредственный доступ и по структуре совпадающий с объектом, к которому предполагается обращаться при выполнении. Его можно рассматривать как конструкцию struct языка C, Java-класс, PL/SQL-запись или структуру данных. Он создается для того, чтобы сервер знал количество и типы столбцов, и другие свойства объекта. Рассмотри это на примере. Предположим, необходимо создать процедуру, обращающуюся к представлению **DBA_USERS** и выдающую в удобном формате оператор **CREATE USER** для любого существующего пользователя. Можно попытаться создать эту процедуру, например, так:

```
tkyte@TKYTE816> create or replace
  2  procedure show_user_info(p_username in varchar2)
  3  AUTHID CURRENT_USER
  4  as
  5      l_rec   dba_users%rowtype;
  6  begin
  7      select *
  8        into l_rec
  9        from dba_users
 10       where username = upper(p_username);
 11
 12     dbms_output.put_line('create user ' || p_username);
 13      if (l_rec.password = 'EXTERNAL') then
 14         dbms_output.put_line(' identified externally');
 15      else
 16          dbms_output.put_line
 17           (' identified by values ''' || l_rec.password || '''');
 18      end if;
 19      dbms_output.put_line
 20      (' temporary tablespace ' || l_rec.temporary_tablespace ||
 21       ' default tablespace ' || l_rec.default_tablespace ||
 22       ' profile ' || l_rec.profile);
 23  exception
 24      when no_data_found then
 25        dbms_output.put_line('*** Нет такого пользователя: ' II
-> p_username) ;
 26  end;
 27  /

Warning: Procedure created with compilation errors.

tkyte@TKYTE816> show err
Errors for PROCEDURE SHOW_USER_INPO:

LINE/COL ERROR


4/13     PLS-00201: identifier 'SYS.DBA_OSERS' must be declared
4/13     PL/SQL: Item ignored
6/5      PL/SQL: SQL Statement ignored
8/12     PLS-00201: identifier 'SYS.DBA_USERS' must be declared
12/5     PL/SQL: Statement ignored
12/10    PLS-00320: the declaration of the type of this expression is
         incomplete or malformed
18/5     PL/SQL: Statement ignored
19/35    PLS-00320: the declaration of the type of this expression is
         incomplete or malformed
```

Эта процедура не компилируется, не потому, что не существует объект **SYS.DBA_USERS,** а потому, что обращаться к **DBA_USERS** мы можем только благодаря предоставленной роли, а в ходе **компиляции** хранимой процедуры роли не используются. Так что же сделать, чтобы эта процедура скомпилировалась? Для этого можно создать собственную таблицу **DBA_USERS.** Это позволит успешно скомпилировать процедуру. Однако по-

скольку это не "реальная" таблица **DBA_USERS,** желаемые результаты при выполнении не будут получены, **пока** мы не выполним процедуру от имени другого пользователя, который может обращаться к реальному представлению **DBA_USERS:**

```
tkyte@TKYTE816>create table dba_users
  2  as
  3  select * from SYS.dba_users where 1=0;

Table created.

tkyte@TKYTE816>alter procedure show_user_info compile;

Procedure altered.

tkyte@TKYTE816> exec show_user_info(USER);
*** Нет такого пользователя TKYTE

PL/SQL procedure successfully completed.

tkyte6TKYTE816>connect system/manager

system@TKYTE816> exec tkyte.show_user_info('TKYTE')
create user TKYTE
identified by values '698F1E51F530CA57'
temporary tablespace TEMP default tablespace DATA profile DEFAULT

PL/SQL procedure successfully completed.
```

Теперь мы получили процедуру, которая, при вызове любым пользователем, кроме создателя, обращается к "правильному" представлению **DBA_USERS** (если вызывающий не имеет права обращаться к **DBA_USERS,** он получит сообщение о том, что таблица или представление не существует). Если же процедуру выполняет создатель, он получает сообщение **"Нет такого пользователя",** поскольку у него объект-шаблон **DBA_USERS** пустой. Все остальные пользователи, однако, получают ожидаемые результаты. Во многих случаях это вполне приемлемо. Например, когда предполагается работа одного и того же кода с разными таблицами. В данном случае, однако, хотелось бы, чтобы эта процедура всегда работала с одним представлением, **DBA_USERS.** Итак, возвращаемся к тому, как обеспечить работу этой процедуры для всех пользователей, включая создателя? Надо использовать объект-шаблон другого типа. Создадим таблицу, структурно совпадающую с представлением **DBA_USERS,** но с другим именем, скажем, **DBA_USERS_TEMPLATE.** Используем эту таблицу для определения типа записи, в которую выбираются данные. После этого мы сможем динамически обращаться к представлению **DBA_USERS** во всех случаях:

```
system@TKYTE816>connect tkyte/tkyte

tkyte@TKYTE816>drop table dba_users;

Table dropped.

tkyte@TKYTE816>create table dba_users_TEMPLATE
  2  as
  3  select * from SYS.dba_users where 1=0,-

Table created.

tkyte@TKYTE816>create or replace
  2 procedure show_user_info(p_username in varchar2)
```

```
  3  AUTHID  CURRENT_USER
  4   as
  5       type rc is ref cursor;
  6
  7      l_rec    dba_users_TEMPLATE%rowtype;
  8      l_cursor rc;
  9   begin
 10       open l_cursor for
 11     'select    *
 12           from dba_users
 13          where username = :x'
 14       USING upper(p_username);
 15
 16       fetch l_cursor into l_rec;
 17       if (l_cursor%found) then
 18
 19           dbms_output.put_line('create user ' || p_username);
 20           if (l_rec.password = 'EXTERNAL') then
 21               dbms_output.put_line(' identified externally');
 22           else
 23               dbms_output.put_line
 24                   (' identified by values ''' || l_rec.password ||''');
 25         end if;
 26           dbms_output.put_line
 27           (' temporary tablespace ' || l_rec.temporary_tablespace ||
 28             ' default tablespace ' || l_rec.default_tablespace ||
 29       '       profile ' || l_rec.profile);
 30       else
 31           dbms_output.put_line( '*** Нет такого пользователя: ' ||
-> p_username );
 32     end if;
 33       close l_cursor;
 34 end;
 35  /

Procedure created.

tkyte@TKYTE816> exec show_user_info(USER);
create user TKYTE
identified by values  '698F1E51F530CA57'
temporary tablespace TEMP default tablespace DATA profile DEFAULT

PL/SQL procedure successfully completed.
```

Итак, в данном случае мы использовали таблицу **DBA_USERS_TEMPLATE** только для того, чтобы упростить создание типа записи, в которую будут выбираться данные. Можно было бы получить описание представления **DBA_USERS** и создать тип записи со всеми соответствующими полями, но мне больше нравится, когда работу за меня делает сервер. После перехода на новую версию Oracle достаточно будет просто пересоздать таблицу-шаблон и процедура перекомпилируется автоматически, при этом все новые/дополнительные столбцы или изменения типов данных будут автоматически учтены.

Проблемы

Как и при использовании любого средства, в работе процедур с правами вызывающего есть ряд нюансов, которые необходимо учитывать. В этом разделе мы попытаемся рассмотреть некоторые из них.

Права вызывающего и использование разделяемого пула

При использовании прав вызывающего для обеспечения доступа одной процедуры к данным в различных схемах, в зависимости от того, кто ее вызывает, следует помнить, что это достигается за счет менее эффективного использования разделяемого пула. При использовании процедур с правами создателя для каждого запроса в процедуре в разделяемом пуле будет не более одного экземпляра соответствующего SQL-оператора. Процедуры с правами создателя максимально эффективно применяют возможности совместного использования SQL-операторов (почему это принципиально важно, см. в главе 10). Процедуры с правами вызывающего подобную эффективность могут не обеспечивать.

Это не хорошо и не плохо. Это просто надо учитывать при задании размера разделяемого пула. При использовании процедур с правами вызывающего мы будем использовать разделяемый пул аналогично тому, как это делает клиент-серверное приложение, использующее интерфейс ODBC или JDBC и непосредственно посылающее серверу операторы ЯМД. Каждый пользователь будет выполнять запрос с одним и тем же текстом, но запросы эти могут различаться. Так что, хотя все выполняют **SELECT * FROM T**, но поскольку таблицы T у всех пользователей разные, для каждого пользователя будет создаваться и помещаться в разделяемый пул отдельный план запроса и другая соответствующая информация. Это необходимо, поскольку каждый раз используется другая таблица T, с другими правами и планами доступа.

Влияние на разделяемый пул легко продемонстрировать на примере. Я создал в одной схеме следующие объекты:

```
tkyte@TKYTE816>create table t (x int);

Table created.

tkyte@TKYTE816>create table t2 (x int);

Table created.

tkyte@TKYTE816>create public synonym T for T;

Synonym created.

tkyte@TKYTE816>create or replace procedure dr_proc
  2  as
  3      l_cnt number;
  4  begin
  5     select count(*) into l_cnt from t DEMO_DR;
  6  end;
  7  /

Procedure created.
```

```
tkyte@TKYTE816> create or replace procedure ir_proc1
  2  authid current_user
  3  as
  4      l_cnt number;
  5  begin
  6     select count(*) into l_cnt from t DEMO_IR_1;
  7  end;
  8  /

Procedure created.

tkyte@TKYTE816> create or replace procedure ir_proc2
  2  authid current_user
  3  as
  4      l_cnt number;
  5  begin
  6     select count(*) into l_cnt from tkyte.t DEMO_IR_2;
  7  end;
  8  /

Procedure created.

tkyte@TKYTE816> create or replace procedure ir_proc3
  2  authid current_user
  3  as
  4      l_cnt number;
  5  begin
  6     select count(*) into l_cnt from t2 DEMO_IR_3;
  7  end;
  8  /

Procedure created.

tkyte@TKYTE816> grant select on t to public;

Grant succeeded.

tkyte@TKYTE816> grant execute on dr_proc to public;

Grant succeeded.

tkyte@TKYTE816> grant execute on ir_proc1 to public;

Grant succeeded.

tkyte@TKYTE816> grant execute on ir_proc2 to public;

Grant succeeded.

tkyte@TKYTE816> grant execute on ir_proc3 to public;

Grant succeeded.
```

Мы создали две таблицы, **T** и **T2**. Существует также общедоступный синоним **T** для таблицы **TKYTE.T**. Все четыре процедуры обращаются либо к таблице **T**, либо к таблице **T2**. Для процедуры с правами создателя, статически связываемой при компиляции, уточнять имя таблицы именем схемы не надо. Процедура с правами вызывающего, IR_**PROC1**, будет обращаться к таблице T через общедоступный синоним. Вторая процедура, **IR_PROC2**, будет использовать полностью уточненную ссылку, а третья процедура, **IR_PROC3**, будет обращаться к T2 без уточнения схемы. Обратите внимание, что

общедоступного синонима для таблицы T2 нет: я намеренно сделал, чтобы процедура **IR_PROC3** при выполнении обращалась к разным таблицам T2.

Затем я создал десять пользователей с помощью следующего сценария:

```
tkyte@TKYTE816> begin
  2     for i in 1 .. 10 loop
  3         begin
  4           execute immediate 'drop user u' || i || ' cascade';
  5         exception
  6             when others then null;
  7         end;
  8         execute immediate 'create user u'||i || ' identified by pw';
  9         execute immediate 'grant create session, create table to u'||i;
 10        execute immediate 'alter user u' || i || ' default tablespace
 11                           data quota unlimited on data';
 12     end loop;
 13  end;
 14  /

PL/SQL procedure successfully completed.
```

и для каждого пользователя мы выполняем:

```
create table t2 (x int);
exec tkyte.dr_proc
exec tkyte.ir_proc1
exec tkyte.ir_proc2
exec tkyte.ir_proc3
```

Необходимо зарегистрироваться как очередной пользователь, создать таблицу T2, а затем выполнить четыре интересующих нас процедуры. После того как это сделано от имени всех десяти пользователей, можно исследовать содержимое разделяемого пула с помощью представления V$SQLAREA, используя представленную ранее в этой главе процедуру PRINT_TABLE:

```
tkyte@TKYTE816> set serveroutput on size 1000000
tkyte@TKYTE816> begin
  2    print_table ('select sql_text, sharable_mem, version_count,
  3                         loaded_versions, parse_calls, optimizer_mode
  4                    from v$sqlarea
  5                   where sql_text like "% DEMO\__R%" escape "\"
  6                   and lower(sql_text) not like "%v$sqlarea%'' ') ;
  7  end;
  8  /
SQL_TEXT                      : SELECT  COUNT(*)  FROM OPS$TKYTE.T
DEMO_IR_2
SHARABLE_MEM                  : 4450
VERSION_COUNT                 : 1
LOADED_VERSIONS               : 1
PARSE_CALLS                   : 10
OPTIMIZER_MODE                : CHOOSE

SQL_TEXT                      : SELECT COUNT(*)   FROM T DEMO_DR
SHARABLE MEM                  : 4246
```

```
VERSION_COUNT                 : 1
LOADED_VERSIONS               : 1
PARSE_CALLS                   : 10
OPTIMIZER_MODE                : CHOOSE

SQL_TEXT                      : SELECT COUNT(*)   FROM T DEMO_IR_1
SHARABLE_MEM                  : 4212
VERSION_COUNT                 : 1
LOADED_VERSIONS               : 1
PARSE_CALLS                   : 10
OPTIMIZER_M0DE                : CHOOSE

SQL_TEXT                      : SELECT COUNT(*)   FROM T2 DEMO_IR_3
SHARABLE_MEM                  : 31941
VERSION_COUNT                 : 10
LOADED_VERSIONS               : 10
PARSE_CALLS                   : 10
OPTIMIZER_MODE                : MULTIPLE CHILDREN PRESENT

PL/SQL procedure successfully completed.
```

Хотя SQL-оператор во всех случаях один и тот же — SELECT **COUNT(*) FROMT2 DEMO_IR_3, — в** разделяемом пуле для него есть десять разных экземпляров кода. Каждому пользователю необходим собственный оптимизированный план выполнения, поскольку запрос ссылается на разные объекты. В тех случаях, когда базовые объекты совпадали и привилегий хватало, планы выполнения SQL-операторов использовались совместно, как и ожидалось.

Итак, если с помощью прав вызывающего вы собираетесь использовать один экземпляр кода для доступа к нескольким различным схемам, необходимо увеличить разделяемый пул, чтобы он вмещал все планы выполнения запросов. Так мы подходим кследующей проблеме.

Производительность

При использовании процедур с правами вызывающего, как вы уже знаете, каждому пользователю может потребоваться отдельный специфический план выполнения запроса. На построение этих дополнительных планов могут потребоваться существенные ресурсы. Анализ запроса — одно из наиболее интенсивно нагружающих процессор действий сервера. "Стоимость" анализа уникальных запросов, который возможен при использовании подпрограмм с правами вызывающего, можно продемонстрировать с помощью утилиты **TKPROF,** показывающей продолжительность анализа операторов. Для выполнения следующего примера необходима привилегия **ALTER SYSTEM:**

```
tkyte@TKYTE816>alter system flush shared_pool;
System altered.
tkyte@TKYTE816>alter system set timed_statistics=true;
System altered.
tkyte@TKYTE816>alter session set sql trace=true;
```

```
Session altered.
tkyte@TKYTE816> declare
  2      type rc is ref cursor;
  3      l_cursor rc;
  4  begin
  5      for i in 1 .. 500 loop
  6          open l_cursor for 'select * from all_objects t' || i;
  7          close l_cursor;
  8      end loop;
  9    end;
 10  /

PL/SQL procedure successfully completed.
```

При этом был выполнен анализ 500 уникальных операторов (в них используются уникальные псевдонимы таблицы). Ситуация аналогична использованию одной процедуры с правами вызывающего 500 пользователями в 500 различных схемах. В итоговом отчете утилиты **TKPROF** для этого сеанса можно найти следующее:

```
OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS
```

call	count	cpu	elapsed	dj.sk	query	current	rows
Parse	1148	17.95	18.03	0	55	15	0
Execute	1229	0.29	0.25	0	0	0	0
Fetch	1013	0.14	0.17	0	2176	0	888
total	**3390**	**18.38**	**18.45**	**0**	**2231**	**15**	**888**

```
Misses in library cache during parse: 536

  504  user  SQL statements in session.
  648  internal SQL statements in session.
 1152  SQL statements in session.
    0  statements EXPLAINed in this session.
```

Теперь выполним блок, не анализирующий уникальный оператор 500 раз:

```
tkyte@TKYTE816> alter system flush shared_pool;

System altered.

tkyte@TKYTE816> alter system set timed_statistics=true;

System altered.

tkyte@TKYTE816> alter session set sql_trace=true;

Session altered.

tkyte@TKYTE816> declare
  2      type rc is ref cursor;
  3      l_cursor rc;
  4  begin
  5      for i in 1 .. 500 loop
  6          open l_cursor for 'select * from all_objects t';
  7          close l_cursor;
```

```
  8      end loop;
  9    end;
 10  /

PL/SQL procedure successfully completed.
```

и в отчете TKPROF увидим:

```
OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS
call     count    cpu    elapsed   disk,  query current    rows

Parse      614   0.74       0.53      1     55       9       0
Execute    671   0.09       0.31      0      0       0       0
Fetch      358   0.08       0.04      8    830       0     272

total     1643   0.91       0.88      9    885       9     272

Misses in library cache during parse: 22

  504  user  SQL statements in session.
  114  internal SQL statements in session.
  618  SQL statements in session.
    0  statements EXPLAINed in this session.
```

Разница **огромна.** Для анализа 500 уникальных операторов (эмулирующих поведение процедуры с правами вызывающего, которая обращается при каждом вызове к другой таблице) требуется 17,95 секунд процессорного времени. Для анализа же 500 одинаковых операторов (эмулирующих использование стандартной процедуры с правами создателя) понадобилось 0,74 секунды процессорного времени. В 24 раза меньше!

Конечно, это надо учитывать. Когда SQL-операторы не используются повторно, система может тратить больше времени на анализ запросов, чем на их фактическое выполнение. Причины этого были рассмотрены в главе 10, посвященной стратегиям и средствам настройки производительности. Там я продемонстрировал необходимость использования связываемых переменных для повторного использования планов запросов.

Однако это не повод отказываться от использования процедур с правами вызывающего. Используйте их, но помните о последствиях.

Более надежный код для обработки ошибок

При создании хранимой процедуры со следующим кодом:

```
begin
   for x in (select pk from t) loop
       update y set c - c+0.5 where d = x.pk;
   end loop;
end;
```

вполне можно быть уверенным, что при отсутствии синтаксических и семантических ошибок **(компилируется** успешно) она будет работать. При использовании процедур с

правами создателя это верно. Я точно знаю, что объекты (таблицы или представления) T и Y существуют, что T доступен для чтения, а **Y** можно изменять.

При использовании процедуры с правами вызывающего, ни в чем нельзя быть уверенным. Существует ли объект T, и если — да, то имеется ли в нем столбец с именем PK? И имею ли я для него привилегию **SELECT**? А если имею, то не через роль ли она получена? Ведь тогда при вызове процедуры из подпрограммы с правами создателя, она не сработает, хотя при непосредственном вызове будет работать прекрасно. Существует ли объект **Y**? И так далее. Другими словами, все условия, которые раньше можно было считать гарантированно выполненными, вызывают сомнению в процедурах с правами вызывающего. Так что, хотя процедуры с правами вызывающего и открывают новые возможности программирования, в некотором отношении они его усложняют.

При использовании представленного выше кода надо готовиться к обработке множества вполне вероятных случаев:

- объекта **T** нет;
- объект **T** есть, но нет необходимых для доступа к нему привилегий;
- объект **T** есть, но в нем нет столбца **PK**;
- объект **T** существует и имеет столбец **PK**, но тип данных столбца отличается от использованного при компиляции;
- все то же в отношении объекта **Y**.

Поскольку изменение объекта **Y** происходит только при получении определенных данных из **T**, мы можем многократно успешно выполнить эту процедуру, но однажды, когда в **T** будут помещены данные, процедура не сработает. Мы никогда не могли обратиться к объекту **Y**, но процедура не сработала потому, что мы впервые "попытались". Ошибка во фрагменте кода произойдет только тогда, когда он выполнится.

Для получения "надежной" процедуры, перехватывающей все возможные ошибки, необходим примерно такой код:

```
create or replace procedure P
authid current_user
as
    no_such_table exception;
    pragma exception_init(no_such_table,-942);
    insufficient_privs exception;
    pragma exception_init(insufficient_privs,-1031);
    invalid_column_name exception;
    pragma exception_init(invalid_column_name,-904);
    inconsistent_datatypes exception;
    pragma exception_init(inconsistent_datatypes,-932);
begin
   for x in (select pk from t) loop
        update y set c = c+0.5 where d = x.pk;
   end loop;
exception
    when NO_SUCH_TABLE then
        dbms_output.put_line('Перехвачена ошибка: ' || sqlerrm);
```

```
    when INSUFFICIENT_PRIVS then
        dbms_output.put_line('Перехвачена ошибка: ' || sqlerrm);
    When INVALID_COLUMN_NAME then
        dbms_output.put_line('Перехвачена ошибка: ' || sqlerrm);
    when INCONSISTENT_DATATYPES then
        dbms_output.put_line('Перехвачена ошибка: ' || sqlerrm);
— ... (дальше идет множество других обработчиков ошибок)...
end;
/
```

Побочные эффекты использования SELECT *

Использование конструкции **SELECT** * в PL/SQL-процедуре с правами вызывающего, обращающейся к разным таблицам при вызове разными пользователями, может быть очень опасно. При этом данные могут быть получены "поврежденными" или в другом порядке. Причина в том, что запись, в которую выполняется выборка данных, настраивается при компиляции, а не при выполнении. Поэтому список столбцов для PL/SQL-объектов (записей) вместо * формируется **при компиляции,** а данные получаются **при выполнении** запроса. Если в другой схеме имеется объект с тем же именем, но с другим порядком столбцов и к нему обращаются из процедуры с правами вызывающего с помощью оператора **SELECT ***, возникает именно такой побочный эффект:

```
tkyte@TKYTE816> create table t (msg varchar2(25), c1 int, c2 int);

Table created.

tkyte@TKYTE816> insert into t values ('c1=1, c2=2', 1, 2);

1 row created.

tkyte@TKYTE816> create or replace procedure P
  2  authid current_user
  3  as
  4  begin
  5      for x in (select * from t) loop
  6                  dbms_output.put_line('msg= ' || x.msg);
  7          dbms_output.put_line('C1         = ' || x.c1);
  8                  dbms_output.put_line('C2 = ' || x.c2);
  9      end loop;
 10 end;
 11 /

Procedure created.

tkyte@TKYTE816> exec p
msg= c1=1, c2=2
C1 = 1
C2 = 2

PL/SQL procedure successfully completed.

tkyte@TKYTE816> grant execute on P to u1;

Grant succeeded.
```

Итак, мы создали процедуру, показывающую содержимое таблицы T. Значение которое она выдает в столбце MSG, я использую, чтобы продемонстрировать предполагаемый ответ. Кроме того, она выдает значения столбцов C1 и C2. Все просто и понятно. Теперь давайте посмотрим, что произойдет, если выполнить процедуру от имени другого пользователя, со своей собственной таблицей T:

```
tkyte@TKYTE816> @connect ul/pw

ul@TKYTE816> drop table t;

Table dropped.

ul@TKYTE816> create table t (msg varchar2(25), c2 int, c1 int);
Table created.

ul@TKYTE816> insert into t values ('c1=2, c2=1', 1, 2);

1 row created.
```

Обратите внимание, что при создании таблицы я изменил порядок столбцов C1 и C2. Здесь я предполагаю, что C1 = 2 и C2 = 1. При выполнении процедуры, однако, получаем следующее:

```
ul@TKYTE816> exec tkyte.p
msg= c1=2, c2=1
C1 = 1
C2 = 2

PL/SQL procedure successfully completed.
```

Не совсем так, как ожидалось, но если вдуматься глубже... При компиляции была автоматически создана неявная запись X. Запись X — это просто структура данных с тремя элементами — **MSG VARCHAR2, C1 NUMBER** и **C2 NUMBE**R. Когда список столбцов **SELECT *** формировался на этапе анализа запроса от имени пользователя **TKYTE**, были получены столбцы **MSG, C1** и **C2** (именно в таком порядке). При выполнении процедуры пользователем **U1** были, однако, получены столбцы MSG, C2 и C1. Поскольку все типы данных совпадают с типами полей неявно созданной записи X, мы не получили сообщения об ошибке **INCONSISTENT DATATYPE** (это тоже могло произойти, если бы типы данных оказались несовместимыми, с точностью до неявного преобразования). Данные были успешно выбраны, но значение столбца C2 помещено в поле записи C1. Это вполне предсказуемый побочный эффект и еще одна причина не использовать операторы SELECT * в производственном коде.

Помните о "скрытых" столбцах

Это очень похоже на представленную ранее проблему **SELECT ***. Она тоже связана с тем, как компилируется PL/SQL-процедура с правами вызывающего и как разрешаются имена и ссылки на объекты. Рассмотрим оператор **UPDATE**, который при выполнении непосредственно в среде SQL*Plus даст другой результат, чем при выполнении в подпрограмме с правами вызывающего. Он работает "правильно" в обеих средах, просто — по-разному.

Когда PL/SQL-код компилируется в базе данных, каждый статический SQL-оператор анализируется и уточняются все идентификаторы. Эти идентификаторы могут быть

именами столбцов или именами PL/SQL-переменных (связываемых переменных). Если это имена столбцов, они оставляются в запросе без изменений. Если же это имена переменных PL/SQL, они заменяются в запросе ссылкой **:BIND_VARIABLE**. Эта замена производится при компиляции, а не при выполнении. Рассмотрим пример:

```
tkyte@TKYTE816> create table t (cl int);
Table created.
tkyte@TKYTE816> insert into t values (1);
1 row created.
tkyte@TKYTE816> create or replace procedure P
  2  authid current_user
  3  as
  4          c2  number default 5;
  5  begin
  6          update t set cl = c2;
  7  end;
  8  /
Procedure created.
tkyte@TKYTE816> exec p
PL/SQL procedure successfully completed.
tkyte@TKYTE816> select * from t;
        C1
         5
tkyte@TKYTE816> grant execute on P to ul;
Grant succeeded.
```

Пока все выглядит нормально. **C1** — это столбец в таблице **T**, а **C2** — переменная PL/SQL. Оператор **UPDATE T SET C1 = C2** обрабатывается сервером при компиляции и преобразуется в **UPDATE T SET C1 = :BIND_VARIABLE**, а значение **:BIND_VARIABLE** передается при выполнении. Теперь, если зарегистрироваться как **U1**, и создать в этой схеме таблицу T:

```
tkyte@TKYTE816> connect ul/pw
ui@TKYTE816> drop table t;
Table dropped.
ul@TKYTE816> create table t (cl int, c2 int);
Table created.
ul@TKYTE816> insert into t values (1, 2);
1 row created.
ul@TKYTE816> exec tkyte.p
PL/SQL procedure successfully completed.
ul@TKYTE816> select * from t;
```

C1	C2
5	2

Это может показаться правильным или неправильным — смотря как к этому подойти. Мы выполнили оператор **UPDATE T SET C1 = C2**, но если бы мы это сделали в командной строке SQL*Plus, то столбец C1 получил бы значение 2, а не 5. Однако, поскольку сервер переписал этот запрос при перекомпиляции так, что в нем не осталось ссылок на **C2**, он делает то же самое с нашим экземпляром T, что и с другим экземпляром **T:** устанавливает столбцу **C1** значение 5. Эта PL/SQL-процедура не может "видеть" столбец **C2**, поскольку **C2** не существует в объекте, с которым она была скомпилирована.

Сначала это кажется странным, поскольку оператора **UPDATE** в переписанном виде мы обычно не видим, но если вы знаете об этом, результат становится вполне объяснимым.

Java и права вызывающего

PL/SQL-процедуры по умолчанию компилируются с правами создателя. Чтобы такая процедура работала с правами вызывающего, надо это специально указать. Java-процедура по умолчанию работает с правами вызывающего. Если необходимо, чтобы она выполнялась с правами создателя, надо явно указать это при загрузке.

В качестве примера я создал таблицу T, такую, что:

```
ops$tkyte@DEV816>create table t (msg varchar2(50));

Table created.

ops$tkyte@DEV816> insert into t values ('Это таблица T, принадлежащая
-> пользователю ' || user);

1 row created.
```

Я также создал и загрузил две хранимые процедуры на Java (для выполнения этого примера необходима привилегия **CREATE PUBLIC SYNONYM).** Эти хранимые процедуры на Java очень похожи на рассмотренные ранее примеры PL/SQL. Они обращаются к таблице T, которая содержит строку, со сведениями о том, кому "принадлежит" эта таблица, и выдают информацию о пользователе сеанса, о текущем пользователе (схеме, определяющей привилегии) и текущей схеме:

```
tkyte@TKYTE816>host type ir_java.java
import java.sql.*;
import oracle.jdbc.driver.*;

public class ir_java
{
public static void test() throws SQLException
  {
  Connection cnx = new OracleDriver().defaultConnection();

  String sql =
      "SELECT MSG, sys_context('userenv','session_user'), "+
                  "sys_context('userenv','current_user'), "+
```

```
                    "sys_context('userenv','current_schema') "+
          "FROM T";

      Statement stmt = cnx.createStatement();
      ResultSet rset = stmt.executeQuery(sql);

      if (rset.next())
         System.out.println( rset.getString(1) +
                             " session_user=" + rset.getString(2)+
                             " current_user=" + rset.getString(3)+
                             " current_schema=" + rset.getString(4));
      rset.close();
      stmt.close();
    }
}
```

```
tkyte@TKYTE816> host dropjava -user tkyte/tkyte ir_java.java

tkyte@TKYTE816> host loadjava -user tkyte/tkyte -synonym -grant u1 -
verbose -resolve ir_java.java
initialization complete
loading  : ir_java
creating : ir_java
resolver :
resolving: ir_java
synonym  : ir_java
```

Представленная выше подпрограмма загружается с правами вызывающего. Теперь загрузим ту же подпрограмму с другим именем. При загрузке подпрограммы с помощью **loadjava** укажем, что она должна выполняться с правами создателя:

```
tkyte@TKYTE816> host type dr_java.java
import java.sql.*;
import oracle.jdbc.driver.*;

public class dr_java
{
... тот же код, что и в предыдущем примере ...
}
tkyte@TKYTE816> host dropjava -user tkyte/tkyte dr_java.java

tkyte@TKYTE816> host loadjava -user tkyte/tkyte -synonym -definer -grant
u1 -verbose -resolve dr_jav
initialization complete
loading  : dr_java
creating : dr_java
resolver :
resolving: dr_java
synonym  : dr_java
```

Итак, отличия между **IR_JAVA** и **DR_JAVA** — имена классов и тот факт, что подпрограмма **DR_JAVA** была загружена с опцией **-definer**.

Затем я создал спецификацию вызова PL/SQL, чтобы можно было выполнять эти процедуры из SQL*Plus. Обратите внимание, что создано четыре версии. Все вызовы хранимых процедур на Java выполняются только через SQL-уровень. Поскольку SQL-уро-

вень, фактически, представляет собой интерфейсную PL/SQL-подпрограмму, в ней тоже можно задавать конструкцию AUTHID. Надо разобраться, что происходит, когда из подпрограммы с правами вызывающего/создателя на уровне PL/SQL вызывается Java-процедура с правами вызывающего/создателя:

```
tkyte@TKYTE816> create OR replace procedure ir_ir_java
  2  authid current_user
  3  as language java name 'ir_java.test()';
  4  /

Procedure created.

tkyte@TKYTE816> grant execute on ir_ir_java to u1;

Grant succeeded.

tkyte0TKYTE816> create OR replace procedure dr_ir_java
  2  as language java name 'ir_java.test()';
  3  /

Procedure created.

tkyte@TKYTE816> grant execute on dr_ir_java to u1;

Grant succeeded.

tkyte@TKYTE816> create OR replace procedure ir_dr_java
  2  authid current_user
  3  as language java name 'dr_java.test()';
  4  /

Procedure created.

tkyte@TKYTE816> grant execute on ir_dr_java to u1;

Grant succeeded.

tkyte@TKYTE816> create OR replace procedure dr_dr_java
  2  authid current_user
  3  as language java name 'dr_java.test()';
  4  /

Procedure created.

tkyte@TKYTE816> grant execute on dr_dr_java to u1;

Grant succeeded.
```

Теперь необходимо создать и наполнить данными таблицу T в схеме TKYTE:

```
tkyte@TKYTE816> drop table t;

Table dropped.

tkyte@TKYTE816> create table t (msg varchar2(50));

Table created.

tkyte@TKYTE816> insert into t values ('Это таблица T, принадлежащая
-> пользователю ' || user);

1 row created.
```

Итак, теперь мы готовы к проверке выполнения от имени пользователя U1, у которого сейчас появится таблица T со строкой, тоже идентифицирующей владельца:

```
tkyte@TKYTE816> @connect u1/pw

u1@TKYTE816> drop table t;

Table dropped.

u1@TKYTE816> create table t (msg varchar2(50));

Table created.

u1@TKYTE816> insert into t values ('Это таблица T, принадлежащая
-> пользователю ' || user);

1 row created.

u1@TKYTE816> set serveroutput on size 1000000
u1@TKYTE816> exec dbms_java.set_output(1000000);

PL/SQL procedure successfully completed.

u1@TKYTE816>exec tkyte.ir_ir_java
Это таблица T, принадлежащая пользователю U1 session_user=U1
current_user=U1 current_schema=U1

PL/SQL procedure successfully completed.
```

Это показывает, что, когда хранимая Java-процедура с правами вызывающего вызывается из PL/SQL-процедуры с правами вызывающего, она работает как процедура с правами вызывающего. Текущий пользователь и текущая схема — U1, SQL-оператор в хранимой Java-процедуре, обращается к таблице U1.T, а не TKYTE.T. Теперь давайте вызовем тот же Java-код из процедуры с правами создателя:

```
u1@TKYTE816>exec tkyte.dr_ir_java
Это таблица T, принадлежащая пользователю TKYTE session_user=U1
current_user=TKYTE current_schema=TKYTE

PL/SQL procedure successfully completed.
```

Теперь, хотя хранимая Java-процедура загружена с правами вызывающего, она работает как процедура с правами создателя. Это вполне можно было предвидеть исходя из того предыдущих примеров. Процедура с правам вызывающего, вызываемая из процедуры с правами создателя, работает аналогично процедуре с правами создателя. Роли не учитываются; текущая схема фиксируется статически, как и текущий пользователь. Эта процедура обращается к таблице **TKYTE.T**, а не к **U1.T**, как в предыдущем примере, а текущий пользователь и схема имеет фиксированное значение — **TKYTE**.

Давайте рассмотрим, что произойдет, если PL/SQL-процедура с правами вызывающего вызывает хранимую Java-процедуру, загруженную с правами создателя:

```
u1@TKYTE816>exec tkyte.ir_dr_java
Это таблица T, принадлежащая пользователю TKYTE session_user=U1
current_user=TKYTE current_schema =TKYTE

PL/SQL procedure successfully completed.
```

Это показывает, что если Java-процедура загружена с опцией **-definer**, она работает с правами создателя, даже если вызывается из процедуры с правами вызывающего. Результат последнего примера теперь уже очевиден. Из PL/SQL-процедуры с правами создателя вызывается Java-процедура с правами создателя:

```
u1@TKYTE816>exec tkyte.dr_dr_java
Это таблица T, принадлежащая пользователю TKYTE session_user=U1
current_user=TKYTE current_schema =TKYTE

PL/SQL procedure successfully completed.
```

Разумеется, она выполняется с правами создателя.

Вы можете даже не заметить, что хранимая Java-процедура загружается с правами вызывающего, поскольку вызывается она обычно из PL/SQL-подпрограмм, а они стандартно компилируются с правами создателя. Обычно Java-код загружается в ту же схему, где создается спецификация вызова; и если она создана с правами создателя, Java-код также работает с правами создателя. Я возьму на себя смелость предположить, что большинство пользователей не подозревает о такой особенности загрузки Java-кода, поскольку по результатам почти никогда нельзя догадаться о работе с правами вызывающего. Только если спецификация вызова на PL/SQL создана с конструкцией **AUTHID CURRENT_USER,** это свойство может проявиться.

Еще одни случай, когда стандартная загрузка Java-кода с правами вызывающего имеет значение, — создание спецификации вызова не в той схеме, в которой загружен байт-код **Java**. Используя тот же загруженный ранее Java-код, я создал от имени пользователя **U1** несколько спецификаций вызова Java-кода из схемы пользователя **TKYTE.** Для этого пользователю U1 предоставлена привилегия **CREATE PROCEDURE.** Кроме того, используется тот факт, что при загрузке Java-кода задана опция **-synonym,** благодаря которой для загруженного кода создан общедоступный синоним, а также опция **-grant U1,** предоставившая пользователю U1 непосредственный доступ к Java-коду. Вот результат:

```
u1@TKYTE816>create OR replace procedure ir_java
  2  authid current_user
  3  as language java name 'ir_java.test()';
  4  /

Procedure created.

u1@TKYTE816>exec ir_java
Это таблица T, принадлежащая пользователю U1 session_user=U1
current_user=U1 current_schema=U1

PL/SQL procedure successfully completed.
```

Вы видите, что процедура с правами вызывающего (процедура с правами создателя дала бы тот же результат), принадлежащая пользователю **U1**, выполняет SQL-операторы в Java-коде так, как если бы ее загрузил пользователь **U1**. Это свидетельствует о том, что Java-код загружен с правами вызывающего. В противном случае, SQL-оператор в Java-коде работал бы с разрешением имен и привилегиями пользователя **TKYTE**, а не **U1**. Следующий пример показывает работу процедуры с правами создателя, созданной в схеме U1. Java-код работает от имени **TKYTE**:

```
u1@TKYTE816> create OR replace procedure dr_java
  2  as language java name 'dr_java.test()';
  3  /

Procedure created.
```

```
ul@TKYTE816>exec dr_java
Это таблица T, принадлежащая пользователю TKYTE session_user=U1
current_user=TKYTE current_schema =TKYTE

PL/SQL procedure successfully completed.
```

Java-код, загруженный с правами создателя, работает от имени **TKYTE,** а не **U1.** Загружать Java-код с правами создателя пришлось принудительно с помощью опции **-definer,** поскольку свойства хранимых процедур на Java отличаются в этом отношении от свойств PL/SQL-процедур.

Возможные ошибки

Других специфических ошибок, кроме тех, что обсуждалось в разделе "Проблемы", при использовании прав вызывающего или создателя не будет. Используя права вызывающего, важно четко понимать, как в **PL/SQL** обрабатываются встроенные SQL-операторы, чтобы избежать проблем с операторами SELECT * и изменяющимся порядком столбцов, "скрытыми" столбцами при выполнении и так далее. Кроме того, работавший без проблем PL/SQL-код при использовании прав вызывающего может выдавать различные ошибки для каждого из пользователей. Причина в том, что ссылки на объекты разрешаются по-разному. В каких-то схемах может не хватать привилегий, использоваться другие типы данных и т.д.

При использовании прав вызывающего необходимо больше внимания уделять надежности кода- и учитывать возможность возникновения ошибок там, где их обычно не бывает. Статические ссылки больше не гарантируют безошибочность работы кода. Ситуация напоминает скорее создание ODBC- или JDBC-программы со встроенными непосредственными вызовами SQL-операторов. Вы контролируете "компоновку" программы (вам известно, какие подпрограммы в клиентском приложении будут вызываться), но вы не контролируете работу SQL-операторов до тех пор, пока они не выполнятся. SQL-оператор в PL/SQL-подпрограмме с правами вызывающего работает также, как в клиентском приложении, использующем интерфейс JDBC. Не проверив все варианты выполнения для каждого пользователя, вы не можете быть на 100 процентов уверены, что в производственной среде все будет работать без ошибок. Поэтому в коде надо больше заботиться об обработке ошибок, чем в традиционной хранимой процедуре.

Резюме

В этой главе мы детально рассмотрели особенности процедур с правами создателя и вызывающего. Вы убедились, как легко обеспечить работу с правами вызывающего, но узнали и то, какой ценой это достигается с точки зрения:

- выявления и обработки ошибок;
- трудно выявляемых ошибок, вызванных иной структурой таблицы при выполнении;
- дополнительного расхода пространства в разделяемой области SQL;
- дополнительных затрат времени на анализ операторов.

Во многих случаях эта цена оказывается неприемлемой. В других же случаях (например, при создании универсальной подпрограммы для выдачи в виде набора строк со значениями столбцов через запятую результатов запроса или при форматировании результатов запроса по вертикали, а не по горизонтали) возможность использовать права вызывающего — бесценна. Без нее просто не удалось бы получить желаемый результат.

Подпрограммы с правами вызывающего имеет смысл применять в таких случаях:

- когда необходимо использовать динамические SQL-операторы (как в упомянутых выше примерах);
- когда выполняемые SQL-операторы обеспечивают защиту на базе идентификатора схемы (как в случае работы со словарем данных или при реализации собственных средств тщательного контроля доступа в приложении);
- когда необходим учет ролей — только подпрограммы с правами вызывающего позволяют этого добиться.

Права вызывающего **можно** использовать для обеспечения доступа к различным схемам на основе текущей схемы, возвращаемой вызовом **SYS_CONTEXT('USERENV', 'CURRENT_SCHEMA'),** но при этом надо позаботиться о согласованности схем и наличии необходимых привилегий (или учитывать возможные проблемы с этим в коде). Надо также быть готовым к более интенсивному использованию разделяемого пула и дополнительному расходу ресурсов сервера на анализ операторов.

Работа с правами создателя идеально подходит почти для всех хранимых процедур. Процедуры с правами вызывающего — мощное средство, но использовать его надо только по назначению.

A

Основные стандартные пакеты

В этом разделе книги будут рассмотрены стандартные пакеты в базе данных, которых, по моему мнению, должен знать каждый. Все эти пакеты описаны в руководстве *Oracle8i Supplied PL/SQL Packages Reference.* В фирменной документации обычно описаны точки входа (общедоступные процедуры и функции) стандартных пакетов и использование каждой функции/процедуры. Я же рассмотрю подробно, когда имеет смысл использовать тот или иной пакет. Не вдаваясь глубоко в работу каждой процедуры, я уделю внимание наиболее часто используемым точкам входа и продемонстрирую, как они используются. Исчерпывающий список содержащихся в каждом пакете процедур и подробное описание параметров вы сможете найти в упомянутом документе.

Освоив это приложение, вы будете хорошо ориентироваться в назначении стандартных пакетов. Мы изучим не все пакеты. Это не означает, что остальные пакеты менее полезны, просто при разработке они используются редко.

Мы рассмотрим пакеты, описанные в приложении.

- **DBMS_ALERT** и **DBMS_PIPE.** Средства межпроцессного взаимодействия в базе данных. Пакет DBMS_ALERT можно использовать для уведомления всех заинтересованных сеансов об определенном событии. Пакет **DBMS_PIPE** позволяет двум сеансам взаимодействовать, аналогично тому, как это происходит через сокет **TCP/IP.**
- **DBMS_APPLICATION_INFO.** Позволяет приложению записать полезную информацию в представления V$. Незаменим в случае контроля действий хранимой процедуры и записи другой информации.
- **DBMS_JAVA.** PL/SQL-пакет, используемый для работы с хранимыми процедурами на языке Java.

- **DBMS_JOB**. Планировщик заданий в базе данных. Используется, если необходимо выполнять хранимую процедуру, например, ежесуточно в 2 часа ночи или просто для выполнения какого-либо действия в фоновом режиме.
- **DBMS_LOB**. Пакет для работы с большими объектами (**L**arge **OB**jects — **LOB**) в базе данных.
- **DBMS_LOCK**. Пакет для создания пользовательских блокировок, независимых от блокировок уровня строки или таблицы, устанавливаемых сервером Oracle.
- **DBMS_LOGMNR**. Пакет для просмотра и анализа содержимого активных журналов повторного выполнения.
- **DBMS_OBFUSCATION_TOOLKIT**. Обеспечивает шифрование данных в базе.
- **DBMS_OUTPUT**. Обеспечивает простые средства вывода информации на экран из PL/SQL для среды SQL*Plus и **SVRMGRL**.
- **DBMS_PROFILER**. Профилировщик исходного кода PL/SQL, встроенный в базу данных.
- **DBMS_UTILITY**. "Сборная солянка" полезных процедур.
- **UTL_FILE**. Обеспечивает средства ввода-вывода текстовых (а в Oracle 9.2.x и двоичных — *прим. научн.ред.)* файлов в PL/SQL. Позволяет читать и записывать текстовые файлы на сервере с помощью PL/SQL.
- **UTL_HTTP**. Обеспечивает работу по протоколу **HTTP** (Hyper Text Transfer Protocol — протокол передачи гипертекста) из среды PL/SQL. Позволяет загружать Web-страницы в PL/SQL.
- **UTL_RAW**. Обеспечивает преобразование данных типа **RAW** в **VARCHAR2**, и наоборот. Используется для работы с протоколом TCP/IP, при обработке больших объектов типа **BLOB** и **BFILE**, а также для шифрования.
- **UTL_SMTP**. Обеспечивает работу по протоколу SMTP (Simple Mail Transfer Protocol — простой протокол передачи электронной почты) из среды PL/SQL. В частности, позволяет послать сообщение по электронной почте из PL/SQL-подпрограммы.
- **UTL_TCP**. Предоставляет средства работы с сокетами TCP/IP в языке PL/SQL. Позволяет из PL/SQL-кода подключиться к любой службе TCP/IP.

Когда используются стандартные пакеты

Причина использования стандартных пакетов проста: гораздо проще разрабатывать и сопровождать код, использующий стандартные средства, чем создавать их самому. Если корпорация Oracle поставляет пакет для определенных целей (например, шифрования данных), не имеет смысла писать такой пакет самому. Часто я сталкиваюсь с тем, что разработчики по незнанию создают средства, уже существующие в базе данных. Знание того, какие готовые инструментальные средства имеются, существенно упрощает разработку.

О стандартных пакетах

Имена всех стандартных пакетов Oracle начинаются с префиксов **DBMS_** или **UTL_**. Имена пакетов, созданных разработчиками серверных технологий (Server Technologies — теми, кто писал ядро базы данных), обычно начинаются с префикса **DBMS_**. Пакеты с именами, начинающимися с **UTL_**, происходят из других источников. В качестве примера можно назвать **UTL_HTTP** — пакет для выполнения HTTP-запросов из PL/SQL (для получения Web-страниц и т.п.). Подразделение по разработке сервера приложений (Application Server Division) корпорации Oracle создало этот пакет для поддержки механизма ICX (Inter-Cartridge eXchange — обмен данными между картриджами) в сервере OAS (Oracle Application Server — сервер приложений Oracle), который сейчас заменен сервером iAS, (internet Application Server — сервер приложений Internet). Это различие имен для разработчиков практического значения не имеет — просто интересно отметить.

Большинство этих пакетов хранится в базе данных в скомпилированном, скрытом (wrapped) формате. Благодаря этому код защищен от любопытных глаз. Можно увидеть спецификацию пакета, но не реализацию. Если выбрать из базы данных код тела пакета DBMS_OUTPUT, будет получено примерно следующее:

```
tkyte@TKYTE816>  select text
  2  from all_source
  3 where name = 'DBMS_OUTPUT'
  4  and type = 'PACKAGE BODY'
  5  and line < 10
  6  order by line
  7  /

TEXT


package body dbms_output wrapped
0
abcd
abcd
abcd
abcd
abcd
abcd
abcd

9 rows selected.
```

Как видите, пользы от этого мало. Можно, однако, выбрать **спецификацию** пакета.*

```
tkyte@TKYTE816>  select text
  2    from all_source
  3   where name = 'DBMS_OUTPUT'
  4     and type = 'PACKAGE'
  5     and line < 26
```

** В базе данных, естественно, все комментарии в спецификации - на английском. Оригинал можно получить, выполнив представленный ниже оператор SELECT. -* **Прим. научн. ред.**

```
  6   order by line
  7  /

TEXT


package dbms_output as


  --  ОБЗОР
  --  Эти процедуры накапливают информацию в буфере  (с помощью "put"
  --  и "put_line") так,  что ее можно выбрать в дальнейшем (с помощью
      "get_line" или "get_lines"). Если пакет отключен, то все вызовы
  --  просто игнорируются. Таким образом,  эти подпрограммы активны,
  --  только если клиент способен обработать получаемую информацию.
  --  Это хорошо подходит для отладки или написания хранимых процедур,
  --  с помощью которых выдаются сообщения или отчеты в среде sql*dba
  --  или plus  (например,  "описательных процедур" и т.п.).
  --  Стандартный размер буфера — 20000. Минимальный — 2000,  а
  --   максимальный -1000000.


  --   ПРИМЕР

  --  Предположим,  из триггера необходимо выдать отладочную информацию.
  --  Для этого в триггере можно вызвать
         dbms_output.put_line('Мы получили:  '||:new.col||
          ' — новое значение');
  --  Если клиент включил пакет dbms_output,  строка-аргумент put_line
  --  будет помещена в буфер и клиент,  выполнив оператор
       (предположительно,  оператор вставки,  удаления или изменения,
  --  вызвавшего срабатывание триггера),  сможет выполнить

25 rows selected.
```

В базе данных скрыт оперативный источник документации. Спецификация каждого из рассматриваемых пакетов содержит достаточно полное описание назначения пакета, действий каждой функции и процедуры, а также их использования. Это очень удобно при отсутствии документации, но и при ее наличии тоже пригодится, поскольку спецификация иногда содержит данные, отсутствующие в документации, или полезные примеры.

Далее мы рассмотрим пакеты, являющиеся большим подспорьем при постоянной работе с СУБД Oracle. Эти пакеты часто используются всеми разработчиками. Мы также рассмотрим новые пакеты и способы обхода некоторых ограничений этих встроенных пакетов, с которыми, как я знаю по своему опыту, часто сталкиваются разработчики.

Пакеты DBMS_ALERT и DBMS_PIPE

Пакеты **DBMS_ALERT** и **DBMS_PIPE** — очень мощные средства межпроцессного взаимодействия. Оба они обеспечивают возможность взаимодействия сеансов базы данных. Пакет **DBMS_ALERT** по функциональности во многом аналогичен *сигналам* операционной системы UNIX, а **DBMS_PIPE** очень похож на *именованный канал* UNIX. Поскольку разработчики приложений часто сомневаются, какой пакет в каких случаях использовать, я решил описать их вместе.

Пакет **DBMS_ALERT** создавался для того, чтобы сеанс мог сигнализировать об определенном событии в базе данных. Другие сеансы, которых касается это событие, получают уведомление о том, что оно произошло. Уведомления эти посылаются в рамках транзакций, т.е. можно сигнализировать о событии в триггере или хранимой процедуре, но пока соответствующая транзакция не зафиксирована, уведомление ожидающим сеансам не посылается. Если транзакция отменена, уведомление не будет послано. Важно понимать, что сеанс, которому необходимо получить уведомление о событии в базе данных, должен либо периодически опрашивать базу данных, не поступил ли соответствующий сигнал, либо просто ждать в заблокированном состоянии возникновения соответствующего события.

Пакет **DBMS_PIPE** более универсален. Он позволяет одному или нескольким сеансам читать сообщения с одной стороны именованного канала и при этом записывать сообщения в этот канал с другой стороны. Только один из читающих сеансов может получить сообщение, причем адресовать сообщение конкретному сеансу по одному именованному каналу нельзя. Если читающих сеансов больше одного, прочитает записанное в канал сообщение любой из них. Каналы не поддерживают транзакции: если сообщение послано, оно будет доступным другим сеансам. Фиксировать транзакцию не

надо, а фиксация или откат соответствующей транзакции не повлияет на результат передачи сообщения по каналу.

Когда использовать сигналы и каналы

Основное отличие между сигналами и каналами — это поддержка (или отсутствие поддержки) транзакций. С помощью сигналов можно передать сообщение одному или нескольким сеансам после того, как некоторое действие успешно зафиксировано в базе данных. Каналы позволяют **немедленно** передать сообщение одному сеансу. Сигналы имеет смысл использовать, например, в следующих случаях.

- На экране в виде графической диаграммы отображаются данные о курсе акций. Когда информация о курсе изменяется в базе данных, необходимо уведомить приложение, чтобы изменить соответственно экран.
- При добавлении новой записи в таблицу необходимо выдать в приложении диалоговое окно, уведомляющее пользователя о новом задании.

Каналы базы данных имеет смысл использовать, например, в следующих случаях.

- На другой машине в сети работает процесс, выполняющий определенные действия. Надо послать процессу сообщение с требованием выполнить необходимое действие. В этом случае канал базы данных используется аналогично сокету TCP/IP.
- Необходимо поставить в очередь в области SGA данные, которые должны быть прочитаны и обработаны другим процессом. При этом канал базы данных используется как непостоянная очередь (FIFO), доступная для чтения нескольким различным сеансам.

Можно привести и другие примеры использования пакетов, но это — наиболее типичные варианты использования сигналов и каналов, позволяющие понять, когда именно надо применять каждый из пакетов. Сигналы используются для уведомления множества пользователей о произошедшем событии (после фиксации). Каналы используются для немедленной передачи сообщения другому сеансу (и, как правило, ожидания от него ответа).

Теперь, когда назначение сигналов и каналов понятно, рассмотрим детали реализации каждого из этих механизмов.

Настройка

Пакеты **DBMS_ALERT** и **DBMS_PIPE** стандартно устанавливаются в базе данных. В отличие от многих стандартных пакетов, привилегия **EXECUTE** для этих пакетов роли **PUBLIC** не предоставляется. В Oracle 8.0 и выше привилегия **EXECUTE** для этих пакетов предоставляется роли **EXECUTE_CATALOG_ROLE**. В предыдущих версиях никакие привилегии на эти пакеты по умолчанию не предоставлялись.

Поскольку привилегия **EXECUTE** предоставлена роли, причем — не **PUBLIC**, вы не сможете создать хранимую процедуру, зависящую от этих пакетов, поскольку при компиляции хранимого кода роли никогда не действуют. Необходимо явно предоставить привилегию **EXECUTE** соответствующей учетной записи.

Пакет DBMS_ALERT

Пакет **DBMS_ALERT** — очень небольшой и содержит всего семь точек входа. Я опишу здесь шесть наиболее интересных. Если для приложения требуется получить уведомление, то наиболее существенными окажутся следующие подпрограммы.

- **REGISTER**. Зарегистрироваться на получение указанного сигнала. В сеансе можно многократно вызывать подпрограмму **REGISTER** с разными именами сигналов, чтобы получать уведомления при наступлении одного из нескольких событий.
- **REMOVE**. Снять регистрацию на получение сигнала, чтобы сервер не пытался уведомить о наступлении события.
- **REMOVEALL**. Снять для сеанса регистрацию на получение всех сигналов.
- **WAITANY**. Ожидать уведомления о сигналах, на получение которых сеанс зарегистрирован. Эта подпрограмма выдает имя поступившего сигнала и обеспечивает доступ к сопровождающему его короткому сообщению. Ждать можно либо заданное время, либо вообще не ждать (что позволяет из приложения эпизодически опрашивать систему, чтобы узнать, не произошло ли событие, не блокируя при этом дальнейшую обработку ожиданием события).
- **WAITONE**. Ожидать уведомления об указанном сигнале. Как и в случае **WAITANY**, ждать можно определенное время или вообще не ждать.

Приложение, желающие послать сигнал, или уведомить о событии, может сделать это с помощью следующей подпрограммы.

- **SIGNAL**. Послать сигнал о наступлении события при фиксации текущей транзакции. При откате посылка сигнала отменяется.

Итак, пакет **DBMS_ALERT** очень просто использовать. Клиентское приложение, для которого требуется получение уведомления о событии, может содержать код вида:

```
tkyte@TKYTE816> begin
  2    dbms_alert.    register('MyMert');
  3 end;
  4 /

PL/SQL procedure successfully completed.

tkyte@TKYTE816>set serveroutput on
tkyte@TKYTE816> declare
  2      l_status    number;
  3      l_msg       varchar2(1800);
  4  begin
  5      dbms_alert.waitone(name    => 'MyMert',
  6                         message => l_msg,
  7                         status  => l_status,
  8                         timeout => dbms_alert.maxwait);
  9
 10     if (l_status = 0)
 11      then
 12     'dbms_output.put_line('Сообщение     события: ' || l_msg);
```

```
13          end if;
14    end;
15    /
```

Мы зарегистрировались на получение сигнала **MyAlert,** а затем вызвали процедуру **DBMS_ALERT.WAITONE,** ожидая поступление этого сигнала. Обратите внимание, что, поскольку использована константа **DBMS_ALERT.MAXWAIT** из пакета **DBMS_ALERT,** сеанс при выполнении этого вызова начнет ждать бесконечно. Сеанс блокируется в ожидании соответствующего события. Можно задать для клиентского приложения период ожидания 0 секунд, что исключит ожидание, и опрашивать сервер о наступлении события. Например, приложение Oracle Forms может использовать ежеминутно срабатывающий таймер, вызывающий процедуру **DBMS_ALERT.WAITONE,** чтобы узнать, не произошло ли некоторое событие. Если событие произошло, экран приложения изменяется. Можно с такой же частотой активизировать поток Java, проверяющий, не произошло ли событие, и обновляющий совместно используемую структуру данных, и т.д.

Чтобы послать этот сигнал, достаточно выполнить следующее:

```
tkyte@TKYTE816> exec dbms_alert.signal('MyMert', 'Hello World');
PL/SQL procedure successfully completed.
tkyte@TKYTE816> commit;
Commit complete.
```

в другом сеансе. В сеансе, заблокированном в ожидании события, вы должны немедленно увидеть:

```
15    /
Сообщение события: Hello World
PL/SQL procedure successfully completed.
```

То есть сеанс больше не заблокирован. Я продемонстрировал наиболее типичный вариант использования пакета **DBMS_ALERT.** Сеансы ждут сигнала с определенным именем. Пока в пославшем сигнал сеансе транзакция не зафиксирована, уведомление о сигнале не посылается. В этом легко убедиться с помощью двух сеансов SQL*Plus.

Работа с сигналами становится более интересной, если задаться следующими вопросами.

- Что происходит, если несколько сигналов более-менее одновременно отправляются разными сеансами?
- Что происходит, если посыпать сигнал несколько раз: сколько сигналов будет сгенерировано в конечном итоге?
- Что происходит, если более одного сеанса пошлет сигнал, после того как я зарегистрировался на его получение, но до вызова одной из процедур ожидания? А что произойдет, если несколько сеансов пошлют сигнал в промежутке между вызовами процедур ожидания?

Ответы на эти вопросы позволят выявить побочные эффекты использования сигналов, которые необходимо учитывать. Я также предложу способы избежать некоторых проблем, связанных со всем изложенным выше.

Одновременные сигналы нескольких сеансов

Если повторно выполнить рассмотренный пример, зарегистрировавшись на получение сигнала **MyAlert** и ожидая его в одном сеансе, а затем запустить два дополнительных сеанса, можно будет увидеть, что произойдет при одновременной передаче сигналов из нескольких сеансов. На этот раз в обоих сеансах мы выполним:

```
tkyte@TKYTE816> exec dbms_alert.signal('MyMert', 'Hello World');
```

(транзакции не фиксируются). Окажется, что сеанс, пославший сигнал вторым, заблокирован. Это показывает, что если N сеансов одновременно пытаются послать один и тот же сигнал, N-1 из них будут заблокированы при вызове **DBMS_ALERT.SIGNAL**. Продолжит работу только один из сеансов. Сигналы должны посылаться последовательно, и следует позаботиться о предотвращении подобных проблем.

Должна обеспечиваться возможность одновременного доступа к базе данных множества сеансов. Пакет **DBMS_ALERT** — одно из тех средств, которое существенно снижает масштабируемость по этому показателю. Если создать для таблицы триггер на событие **INSERT**, а в коде этого триггера использовать вызов **DBMS_ALERT.SIGNAL**, при выполнении множества операторов **INSERT** все они выстроятся в очередь, если хоть один сеанс зарегистрировался на получение соответствующего сигнала. Поэтому имеет смысл ограничить количество сеансов, посылающих сигналы. Например, при получении оперативных данных из внешнего источника пакет **DBMS_ALERT** вполне можно использовать, если данные в таблицу вставляет только один сеанс. Если же речь идет о таблице проверки, в которую часто вставляют данные все сеансы, средства пакета **DBMS_ALERT** лучше не использовать.

Один из способов избежать выстраивания сеансов в очередь — использовать пакет **DBMS_JOB** (которому посвящен специальный раздел в этом приложении). Можно написать процедуру, действия которой будут сводиться к передаче сигнала и фиксации транзакции:

```
tkyte@TKYTE816> create table alert_messages
  2  (job_id     int primary key,
  3   alert_name varchar2(30),
  4   message    varchar2(2000)
  5  )
  6  /

Table created.

tkyte@TKYTE816> create or replace procedure background_alert(p_job in int)
  2  as
  3      l_rec alert_messages%rowtype;
  4  begin
  5     select * into l_rec from alert_messages where job_id = p_job;
  6
  7      dbms_alert.signal(l_rec.alert_name, l_rec.message);
  8      delete from alert_messages where job_id = p_job;
  9      commit;
 10  end;
 11  /

Procedure created.
```

Тогда соответствующий триггер будет иметь вид:

```
tkyte@TKYTE816>create table t (x int);
Table created.

tkyte@TKYTE816>create or replace trigger t_trigger
  2  after insert or update of x on t for each row
  3  declare
  4          l_job number;
  5  begin
  6        dbms_job.submit(l_job, 'background_alert(JOB);');
  7          insert into alert_messages
  8          (job_id, alert_name, message)
  9           values
 10            (l_job, 'MyAlert', 'X в T имеет значение ' || :new.x);
 11  end;
 12  /

Trigger created.
```

И будет обеспечивать передачу сигнала фоновым процессом **после** фиксации. При этом:

- сигналы посылаются в рамках транзакций;
- приоритетные процессы (интерактивные приложения) в очередь не выстраиваются.

Недостаток этого подхода в том, что задания могут выполняться не сразу; может пройти некоторое время, прежде чем сигнал будет послан. Во многих случаях это приемлемо (важно уведомить ожидающие процессы о произошедшем событии, даже если и с небольшим опозданием). *Средства расширенной поддержки очередей* (advanced queues — AQ) также обеспечивают хорошо масштабируемый способ уведомления о событиях в базе данных. Использовать их сложнее, чем средства пакета **DBMS_ALERT,** но при этом обеспечивается большая гибкость.

Неоднократная передача сигнала в сеансе

Теперь попытаемся ответить на вопрос, что произойдет, если послать одноименный сигнал в приложении несколько раз, а затем зафиксировать транзакцию? Сколько сигналов фактически будет послано? В данном случае ответ простой: один. Работа пакета **DBMS_ALERT** аналогична механизму передачи сигналов в ОС UNIX. В UNIX сигналы посылаются для уведомления процессов о событиях в операционной системе. Пример такого события — **'I/O is ready'** (готовность к вводу-выводу), которое означает, что один из открытых файлов (или сокетов и т.п.) готов для продолжения операций ввода-вывода. Этот сигнал можно, например, использовать при создании сервера на основе протоколов TCP/IP. Операционная система уведомит вас, когда в одном из открытых сокетов появятся данные, ожидающие чтения; то есть не придется постоянно опрашивать состояние каждого сокета, проверяя, нет ли в нем данных для чтения. Если ОС пять раз определила, что в сокете есть данные для чтения, но у нее не было возможности уведомить об этом приложение, она не будет повторять сообщение пять раз. Вы полу-

чите уведомление о событии "из сокета X можно читать", но не всю хронологию предыдущих событий по этому сокету. Пакет **DBMS_ALERT** работает точно так же.

Возвращаясь к рассматриваемому примеру, можно выполнить фрагмент кода, регистрирующий сеанс на уведомление о событии и вызывающий процедуру **WAITONE** в ожидании этого события. В другом сеансе выполняем:

```
tkyte@TKYTE816> begin
  2   for i in 1 .. 10 loop
  3       dbms_alert.signal('MyAlert', 'Сообщение ' || i);
  4   end loop;
  5  end;
  6  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> commit;

Commit complete.
```

И в первом окне получаем результат:

```
Сообщение события: сообщение 10

PL/SQL procedure successfully completed.
```

Послано будет только последнее сообщение, о котором мы сигнализировали. Промежуточных сообщений никто никогда не увидит. Следует учитывать, что пакет **DBMS_ALERT** будет, как и задумано создателями, отбрасывать все предыдущие сообщения сеанса. С помощью этого пакета нельзя отправить в транзакции последовательность сообщений — это только механизм сигнализации. Он позволяет уведомить клиентское приложение, что "нечто произошло". Если вы предполагаете, что каждое событие, о котором вы уведомляли с помощью сигнала, будет получено всеми заинтересованными сеансами, — вас ждет разочарование (написанный на основе этого предположения код будет скорее всего ошибочен).

Для решения этой проблемы можно использовать пакет **DBMS_JOB,** если для уведомления о каждом событии применять его средства. Однако можно использовать и другую технологию. С помощью средств расширенной поддержки очередей (которые в этой книге не рассматриваются) справиться с этой задачей намного проще.

Передача многочисленных сигналов несколькими сеансами до вызова процедуры ожидания

Это последний вопрос: что произойдет, если сигнал будет послан несколькими сеансами после того, как на него поступил запрос, но прежде, чем вызвана процедура ожидания? Аналогичный вопрос: что произойдет, если между вызовами процедур ожидания несколько сеансов пошлют сигнал? Как и в случае многократного вызова **DBMS_ALERT.SIGNAL** в одном сеансе, запоминается только **последний** сигнал, и именно о нем получат уведомление сеансы. В этом можно убедиться, добавив команду **PAUSE** к используемому в примерах сценарию SQL*Plus:

```
begin
    dbms_alert.register('MyAlert');
end;
/
pause
```

Затем в других сеансах вызовите процедуры **DBMS_ALERT.SIGNAL** с уникальными сообщениями (чтобы их можно было различать) и зафиксируйте каждое сообщение. Например, измените представленный ранее простой цикл следующим образом:

```
tkyte@TKYTE816> begin
  2    for i in 1 .. 10 loop
  3        dbms_alert.signal('MyMert', 'Сообщение ' || i);
  4        commit;
  5    end loop;
  6  end;
  7  /

PL/SQL procedure successfully completed.
```

После этого в исходном сеансе просто нажмите клавишу **Enter,** и блок кода, вызывающий процедуру **WAITONE,** будет выполнен. Поскольку ожидаемый сигнал уже послан, этот блок кода немедленно завершит работу и выдаст строку, свидетельствующую о получении **последнего** сообщения (о чем оповестил сигнал). Все промежуточные сообщения других сеансов потеряны, как и было задумано создателями пакета.

Итак, пакет **DBMS_ALERT** подходит для тех случаев, когда необходимо уведомить о событиях в базе данных множество клиентов. Об этих именованных событиях должно сообщать как можно меньше сеансов, из-за существенных проблем с очередностью доступа к процедурам пакета **DBMS_ALERT.** Поскольку неоднократные сообщения теряются, пакет **DBMS_ALERT** подходит в качестве средства уведомления о **событии.** Его можно использовать для уведомления клиента, например, об изменении данных в таблице Т, но попытка использовать его для уведомления об изменениях в отдельных строках таблицы Т закончится неудачей (поскольку сохраняется только последнее сообщение). Пакет **DBMS_ALERT** очень прост в использовании и практически не требует настройки.

Пакет DBMS_PIPE

DBMS_PIPE — это стандартный пакет, обеспечивающий обмен данными между двумя сеансами. Это средство межпроцессного взаимодействия. Один сеанс может записывать сообщение в программный канал, а другой — читать это сообщение. В ОС UNIX аналогичный механизм реализован в виде именованного канала операционной системы. С помощью именованных каналов можно обеспечить запись данных одним процессом для другого.

Пакет **DBMS_PIPE,** в отличие от **DBMS_ALERT,** — это пакет, работающий в режиме реального времени. При вызове функции **SEND_MESSAGE** немедленно посылается сообщение. Сервер не ждет выполнения оператора **COMMIT;** передача сообщения выполняется вне транзакции. Это позволяет использовать пакет **DBMS_PIPE** в тех случаях, когда **DBMS_ALERT** не подходит (и наоборот). С помощью пакета

DBMS_PIPE можно обеспечить диалоговое взаимодействие двух сеансов (что с помощью **DBMS_ALERT** сделать невозможно). Один сеанс может "попросить" другой выполнить некоторое действие. Выполнив его, второй сеанс возвращает первому результат. Предположим, второй сеанс — это C-программа, которая снимает показания термометра, подключенного к последовательному порту компьютера, и возвращает значение температуры первому сеансу. Первому сеансу надо записать текущую температуру в базу данных. Он может послать сообщение "дай мне значение температуры" второму сеансу, который определяет это значение и выдает ответ первому сеансу. Первый и второй сеансы могут работать на разных компьютерах, главное, оба они подключены к одной базе данных. Я использую базу данных примерно так же, как сеть TCP/IP — для обеспечения взаимодействия между двумя процессами. Однако при использовании пакета **DBMS_PIPE** мне не нужно знать имя хоста и номер порта для подключения — достаточно имени канала базы данных, в который надо отправить запрос.

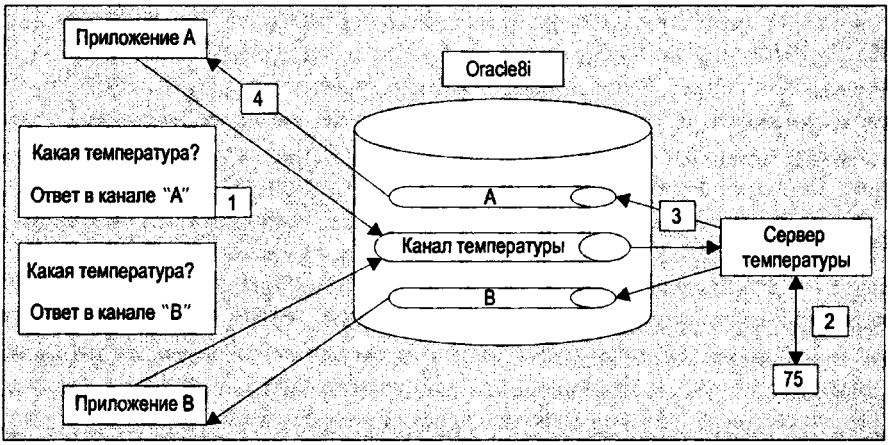
В базе данных есть два типа каналов — *общедоступные* и *пользовательские*. Общедоступный канал можно создать явно, вызвав **CREATE_PIPE,** либо неявно, послав в него сообщение. Основное отличие между явно и неявно созданными каналами состоит в том, что канал, созданный явным вызовом **CREATE_PIPE,** удаляется приложением по завершении работы, тогда как неявно созданный канал удаляется из области SGA как устаревший после определенного промежутка времени. Общедоступный канал устроен так, что **любой** сеанс, имеющий доступ к пакету **DBMS_PIPE,** может читать и записывать в него сообщения. Поэтому общедоступные каналы не подходят для передачи секретных или просто важных данных. Поскольку каналы обычно используются для диалога, а общедоступные каналы позволяют перехватывать или вмешиваться в этот диалог любому, злонамеренный пользователь может удалять сообщения из канала либо добавлять свои, "мусорные". Любое из этих действий нарушает диалог или протокол обмена данными между сеансами. Поэтому в большинстве приложений применяются пользовательские каналы.

К данным в пользовательских каналах можно обращаться только сеансам, работающим с эффективным идентификатором пользователя-владельца канала, или от имени специальных пользователей **SYS** и **INTERNAL.** Это означает, что **только** с помощью хранимых процедур с правами создателя (см. главу 23, посвященную правам создателя и вызывающего), принадлежащих владельцу канала, либо в сеансах от имени владельца канала, пользователя **SYS** или **INTERNAL** можно читать или записывать данные в этот канал. Это существенно увеличивает надежность каналов, поскольку ни один другой сеанс или код не может вмешаться в протокол или перехватить данные.

Канал — это объект в области SGA экземпляра Oracle. Этот механизм вообще не связан с диском. Данные в каналах теряются при остановке и перезапуске сервера.

Чаще всего каналы используют для создания специализированных служб или серверов. До появления внешних процедур в Oracle 8.0 они были единственным способом реализовать хранимые процедуры на языке, отличном от PL/SQL. Для этого создавался "канальный" сервер. Компонент ConText (предшественник компонента interMedia Text) был реализован в Oracle 7.3 с помощью каналов базы данных и с тех пор так и работает. Со временем часть его функций была реализована с помощью внешних процедур, но большая часть механизмов индексирования по-прежнему реализуется с помощью каналов.

Поскольку делать попытку читать данные из канала и писать их туда может любое количество сеансов, необходимо реализовать алгоритм, гарантирующий доставку сообщений нужному сеансу. Если предполагается создание специализированной службы (например, представленного ранее сервера температуры) и ее добавление в базу данных, необходимо гарантировать получение ответа, предназначенного сеансу А, именно сеансом А, а не сеансом В. Для удовлетворения этого стандартного требования обычно запрос выдается в один канал с общеизвестным именем, а вместе с обращением передается уникальное имя канала, из которого мы хотим прочитать ответ. Это можно проиллюстрировать следующей схемой:



Шаг 1. Сеанс А пишет свое обращение — "Какая сейчас температура? Ответить» канал А" — в известный всем сеансам "канал температуры". Одновременно это могут делать и другие сеансы. Каждое сообщение помещается в канал по принципу очереди — "первым пришел, первым обслужен".

Шаг 2. Сервер температуры читает одно сообщение из канала и запрашивает соответствующую службу, к которой он обеспечивает доступ.

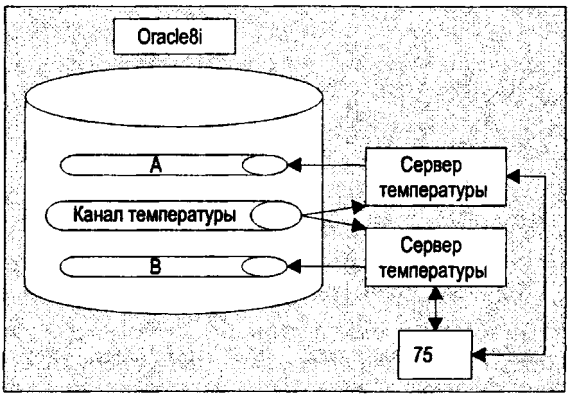
Шаг 3. Сервер температуры использует уникальное имя канала, которое запрашивающий сеанс (в данном случае канал А) указал в сообщении, для записи ответа. Для ответа используется неявный канал (так что канал ответа исчезает сразу после того, как работа с ним завершена). Если подобных вызовов предполагается много, имеет смысл использовать явно созданный канал, чтобы он сохранялся в области SGA i течение всего сеанса (не забудьте удалить его перед завершением сеанса!).

Шаг 4. Сеанс А читает ответ из канала, который он указал серверу температуры для записи.

Такая же последовательность событий произойдет и для сеанса В. Сервер температуры будет читать обращение, запрашивать температуру, определять по запросу канал, в который надо выдать ответ, и записывать ответ в него.

Одна из интересных особенностей каналов базы данных — возможность читать из канала **несколькими** сеансами. Помещенное в канал сообщение будет прочитано только

одним сеансом, но читать сообщения из канала может несколько сеансов одновременно. Это позволяет масштабировать представленную выше схему. По ней понятно, что запрашивать данные у сервера температуры может несколько сеансов, и он будет последовательно, по одному, обрабатывать эти запросы. Но ничего не мешает запустить несколько серверов температуры:



Теперь можно обрабатывать два запроса одновременно. Если запустить пять экземпляров сервера, обрабатывать можно одновременно пять запросов. Это похоже на пул подключений или на работу многопотокового сервера Oracle. Имеется пул процессов, готовых к работе, и максимальный объем одновременно выполняемых действий в каждый момент времени определяется количеством запущенных процессов. Эта особенность каналов базы данных позволяет легко масштабировать систему.

Серверы каналов или внешние процедуры?

В Oracle8 версии 8.0 впервые появилась возможность реализовать хранимые процедуры непосредственно на языке C, а сервер Oracle8i позволил создавать их также на языке Java. С учетом этого, нужны ли теперь пакет **DBMS_PIPE** и серверы каналов? Однозначный ответ: да.

В главе, посвященной использованию внешних процедур, была описана соответствующая архитектура. Внешние процедуры на языке C, например, выполняются в отдельном адресном пространстве по отношению к хранимой процедуре на PL/SQL. Имеется однозначное соответствие между количеством сеансов, одновременно использующих внешние процедуры, и количеством созданных отдельных адресных пространств. Т.е., если 50 сеансов одновременно вызовут внешнюю процедуру, будет запущено 50 процессов или, по крайней мере, потоков **EXTPROC.** Механизм поддержки внешних процедур на языке C похож на механизм работы выделенного сервера Oracle. Сервер Oracle запускает отдельный серверный процесс для каждого из одновременно работающих сеансов; точно так же он запускает экземпляр **EXTPROC** для каждого из одновременных вызовов внешних подпрограмм. Внешние процедуры на Java выполняются аналогично: с соответствием один к одному. Для каждого сеанса, использующего внешнюю процедуру на Java, запускается отдельный экземпляр **JVM**, с собственной информацией о состоянии и специально выделенными ресурсами.

Сервер каналов, с другой стороны, подобен многопотоковому серверу (MTS) в Oracle. Вы создаете пул разделяемых ресурсов (запускаете N серверов каналов), и они обслуживают обращения. Если одновременно поступает больше обращений, чем можно обработать, они выстраиваются в очередь. Это очень похоже на многопотоковый режим работы сервера Oracle, когда обращения поступают в очередь в области SGA и выбираются из очереди разделяемым сервером после обработки им предыдущего обращения. Это хорошо демонстрирует ранее рассмотренный пример сервера температуры. На первой схеме показан один сервер канала; он определяет и выдает температуру клиентам по одному. На второй схеме представлены два сервера канала, обслуживающие все поступающие обращения. Одновременно к термометру будут обращаться не более двух клиентов.

Это важно потому, что позволяет ограничить одновременный доступ к этому разделяемому ресурсу. Если бы использовались внешние процедуры и температуру запросили бы одновременно 50 сеансов, они могли бы "разрушить" программное обеспечение термометра, если оно не способно поддерживать такое количество одновременных обращений. Замените термометр любым другим разделяемым ресурсом, и возникнут такие же проблемы. Можно допустить несколько одновременных обращений, но если их будет много, то либо произойдет сбой, либо производительность снизится настолько, что практически ресурс работать перестанет.

Еще одна причина использования сервера каналов может быть связана с тем, что для подключения к разделяемому ресурсу требуется много времени. Например, пару лет назад я работал над проектом в большом университете. Требовалось выполнять транзакции на мэйнфрейме (необходимо было обращаться к мэйнфрейму для получения информации о студентах). Подключение к мэйнфрейму могло потребовать от 30 до 60 секунд, но после этого информация выдавалась очень быстро (если только мы не перегружали мэйнфрейм огромным количеством одновременных обращений). Используя сервер каналов, мы смогли подключаться к серверу **один раз,** при запуске сервера каналов. Сервер каналов работал много дней, используя первоначальное подключение. Если бы использовалась внешняя процедура, пришлось бы инициировать подключение для каждого сеанса. Реализация на основе внешних процедур в этой среде просто не работала бы из-за продолжительности подключения к мэйнфрейму. Сервер каналов не только позволил ограничить количество одновременных обращений к мэйнфрейму, но и выполнять продолжительный процесс подключения **один раз,** а затем использовать это подключение сотни тысяч раз.

Если вы знакомы с обоснованием использования программного обеспечения для организации пула подключений в трехкомпонентной среде, вам и так понятно, зачем могут понадобиться каналы. Они позволяют повторно использовать результаты выполнения продолжительной операции (подключения к базе данных, в случае ПО для организации пула подключений) и ограничить объем потребляемых одновременно ресурсов (размер пула подключений).

Последнее отличие сервера каналов от внешних процедур связано с тем, где может работать сервер каналов. Предположим, в случае сервера температуры сервер баз данных работает на платформе Windows. Контроль температуры осуществляется на UNIX-машине. При этом все доступные серверу библиотеки тоже находятся в ОС UNIX. Поскольку сервер каналов — всего лишь клиент базы данных, его можно запрограммировать,

скомпилировать и запустить в среде UNIX. Сервер каналов необязательно должен работать на той же машине и даже аппаратной платформе, что и сервер баз данных. Внешняя же процедура должна выполняться на той же машине, что и сервер базы данных, — такие процедуры нельзя выполнять на удаленной машине. Поэтому сервер каналов можно использовать в ситуациях, когда внешнюю процедуру использовать невозможно.

Пример в сети Internet

На Web-сайте издательства Wrox **(http://www.wrox.com)** можно найти пример небольшого сервера каналов. Он отвечает на часто задаваемый вопрос: как выполнить команду базовой операционной системы из PL/SQL? После добавления поддержки Java и внешних процедур на языке C, такую функцию выполнения команд можно легко реализовать с помощью любой из этих технологий. А если компилятора языка C просто нет или поддержка Java в базе данных не установлена — что тогда? Этот пример показывает, как создать небольшой "сервер каналов", способный выполнять команды операционной системы, используя только утилиту SQL*Plus и командный интерпретатор **csh.** Сервер получился сравнительно простым, содержащим лишь несколько строк кода командного интерпретатора **csh** и еще меньше текста на языке PL/SQL. Однако он показывает основные возможности каналов базы данных и должен натолкнуть вас на идеи по созданию других полезных приложений.

Резюме

Каналы базы данных — мощное средство сервера Oracle, позволяющее двум сеансам вести диалог. Созданные по аналогии с именованными каналами в ОС UNIX, они позволяют разработать собственный протокол пересылки и получения сообщений. Небольшой пример, представленный на сайте издательства Wrox, демонстрирует, как создать "сервер каналов" — внешний процесс, получающий обращения от сеансов базы данных и выполняющий для них кое-какие специальные действия. Каналы базы данных работают вне транзакций, что отличает их от сигналов, но именно это делает их во многих случаях незаменимыми. Я использовал каналы базы данных, в частности, для реализации следующих возможностей:

- передачи сообщений электронной почты;
- распечатывания файлов;
- интеграции с другими источниками данных, находящимися вне баз данных Oracle и не поддерживающими язык SQL;
- реализации аналога процедуры **DBMS_LOB.LOADFROMFILE** для типов данных **LONG** и **LONG RAW.**

Пакет DBMS_APPLICATION_INFO

Это — один из наименее используемых стандартных пакетов, хотя его функциональные возможности пригодятся в любом приложении. Интересовались ли вы когда-нибудь следующими вопросами:

- Что делает сеанс, какая форма обрабатывается, какой код выполняется в модуле?
- Насколько близко к завершению выполнение хранимой процедуры?
- Насколько близко к завершению выполнение пакетного задания?
- Какие значения связываемых переменных использованы в запросе?

Пакет **DBMS_APPLICATION_INFO** можно использовать для получения ответов на эти и многие другие вопросы. Он позволяет устанавливать значения трех столбцов — **CLIENT_INFO**, **ACTION** и **MODULE** — соответствующей сеансу строки в представлении **V$SESSION**. Пакет предоставляет функции не только для установки этих значений, но и для их получения. Кроме того, один из параметров встроенных функций **USERENV** и **SYS_CONTEXT** позволяет получить значения столбца **CLIENT_INFO** в запросе. Можно, например, выполнить **SELECT USERENV('CLIENT_INFO') FROM DUAL** или использовать конструкцию **WHERE SOME_COLUMN = SYS_CONTEXT('USERENV','CLIENT_INFO')** в запросах. Значения, устанавливаемые в представлениях **V$**, сразу же доступны в других сеансах. Фиксировать их не нужно, что позволяет эффективно использовать эти представления для взаимодействия с "внешним миром". Наконец, пакет **DBMS_APPLICATION_INFO** позволяет задать значения в представлении динамической производительности **V$SESSION_LONGOPS** (**LONG OP**eration**S**). Это удобно для записи сделанного в продолжительных заданиях.

Многие инструментальные средства Oracle, например SQL*Plus, уже используют возможности этого пакета. Я создал сценарий **SHOWSQL.SQL,** позволяющий узнать, какие SQL-операторы пользователи выполняют в настоящий момент в базе данных (этот сценарий можно найти на Web-сайте издательства Wrox, **http://www.wrox.com).** Часть этого сценария выбирает данные из представления **V$SESSION**, в которых значения столбцов **CLIENT_INFO, MODULE** или **ACTION** непустые (NOT **NULL).** При запуске этого сценария я получаю, например, следующие результаты:

```
USERNAME                  MODULE          ACTION          CLIENT_INFO

OPS$TKYTE(107,19225)   01@ showsql.sql
OPS$TKYTE(22,50901)    SQL*Plus
```

Первая строка показывает, что текущий сеанс выполняет сценарий **SHOWSQL.SQL** на уровне 01. Это означает, что сценарий вызван непосредственно, а не из другого сценария. Если создать сценарий **TEST.SQL** с единственной строкой **@SHOWSQL,** то для представления этого вложенного вызова утилита SQL*Plus установит для сценария **SHOWSQL** уровень вызова 02. Вторая строка показывает другой сеанс SQL*Plus. В нем сейчас никакие сценарии не выполняются (возможно, в нем выполняется команда, введенная непосредственно в командной строке). Если добавить соответствующие вызовы процедур пакета **DBMS_APPLICATION_INFO** в приложение, можно добиться такого же результата, расширив возможности контроля работы приложения для его создателя и администратора базы данных.

Для установки соответствующих значений в представлении V$SESSION используются следующие вызовы.

- **SET_MODULE**. Эта процедура позволяет установить в представлении **V$SESSION** значения столбцов **MODULE** и **ACTION.** Имя модуля должно быть не длиннее 48 байт, а значение поля, описывающего действие, не должно быть длиннее 32 байт. В качестве имени модуля обычно указывается имя приложения. Первым действием можно указать **STARTUP** или **INITIALIZING,** чтобы отметить начало работы приложения.
- **SET_ACTION.** Эта процедура позволяет установить в представлении **V$SESSION** значение столбца **ACTION** (действие). Это значение должно быть таким, чтобы можно было понять, какая часть кода программы выполняется. В качестве действия можно указывать, например, имя текущей активной экранной формы, имя функции в программе на Pro*C или имя PL/SQL-подпрограммы.
- **SET_CLIENT_INFO.** Эта процедура позволяет сохранить до 64 байт специфической информации о приложении, которая может понадобиться. Обычно (см. далее) так сохраняются параметры представления или запроса.

Имеются процедуры и для чтения соответствующей информации из представлений. Помимо установки значений в представлении **V$SESSION** этот пакет позволяет устанавливать значения в представлении динамической производительности **V$SESSION_LONGOPS.** Это представление позволяет сохранять несколько строк информации в различных столбцах. Вскоре мы более подробно рассмотрим эту возможность.

Использование информации о клиенте

Процедура **SET_CLIENT_INFO** позволяет установить не только значения в столбцах представления **V$SESSION,** но и значение переменной **CLIENT_INFO,** которое можно получить с помощью встроенных функций **userenv** (в версиях Oracle 7.3 и выше) или **sys_context** (именно ее лучше использовать в версиях Oracle 8i и выше). Например, можно создать **параметризованное представление,** результаты выборки данных из которого зависят от значения переменной **CLIENT_INFO.** Эту идею можно проиллюстрировать следующим примером:

```
scott@TKYTE816> exec dbms_application_info.set_client_info('KING');
PL/SQL procedure successfully completed.
scott@TKYTE816> select userenv('CLIENT_INFO')  from dual;
USERENV('CLIENT_INFO')

KING
scottTKYTE816> select sys_context('userenv','client_info')from dual;
SYS_CONTEXT('USERENV','CLIENT_INFO')

KING
scott6TKYTE816> create or replace view
  2  emp_view
  3  as
  4  select ename, empno
  5    from emp
  6   where ename = sys_context('userenv',  'client_info');
View created.
scott@TKYTE816> select * from emp_view;
ENAME           EMPNO


KING             7839
scott@TKYTE816> exec dbms_application_info.set_client_info('BLAKE');
PL/SQL procedure successfully completed.
scotteTKYTE816> select * from emp_view;
ENAME           EMPNO

BLAKE            7698
```

Как видите, можно установить соответствующее значение и легко использовать его в запросах вместо константы. Это позволяет создавать сложные представления с условиями, значения которых уточняются при выполнении. При использовании представлений могут возникать проблемы, связанные с *включением условия* (predicate merging). Если оптимизатор включает условие в определение представления, запрос из представления выполняется очень быстро. В противном случае запрос выполняется медленно. Используя значение переменной **CLIENT INFO,** можно включить условие заранее, если

оптимизатор не может этого сделать. Разработчик приложения должен установить переменной соответствующее значение и выполнить SELECT * из представления. В результате он получит нужные данные.

Средства пакета DBMS_APPLICATION_INFO я применяю также для записи значений связываемых переменных, используемых в запросе и другой необходимой информации, позволяющей легко понять, что именно делают процедуры. Например, если включить в долго выполняющийся процесс следующие вызовы:

```
tkyte@TKYTE816> declare
  2     l_owner varchar2(30) default 'SYS';
  3      l_cnt   number default 0;
  4  begin
  5       dbms_application_info.set_client_info('owner-'||l_owner);
  6
  7      for x in (select * from all_objects where owner — l_owner)
  8      loop
  9          l_cnt := l_cnt+1;
 10      dbms_application_info.set_action('processingrow      ' || l_cnt);
 11      end loop;
 12  end;
 13  /
```

то с помощью сценария SHOWSQL.SQL можно получить такую информацию:

```
tkyte@TKYTE816> @showsql
USERNAME                              SID   SERIAL#   PROCESS    STATUS

TKYTE                                   8       206  780:716    ACTIVE
TKYTE                                  11       635  1004:1144  ACTIVE

TKYTE(11,635) ospid = 1004:1144 program = SQLPLUS.EXE
Saturday  15:59  Saturday  16:15
SELECT *   FROM ALL_OBJECTS  WHERE OWNER = :b1

USERNAME        MODULE          ACTION          CLIENT_INFO

TKYTE(8,206)   01@showsql.sql
TKYTE(11,635)   SQL*Plus      processingrow  owner=SYS
                               5393
```

Сеанс (11,635) выполняет запрос **SELECT * FROM ALL_OBJECTS WHERE** OWNER = :B1. Отчет также показывает, что запрос выполнен пользователем **SYS** (owner=SYS) и что в момент вызова сценария showsql он уже обработал S393 строки. В следующем разделе мы увидим, как с помощью представления **V$SESSION_LONGOPS** получить еще более точную информацию, если заранее известно, сколько действий или шагов будет выполнять процедура.

Использование представления V$SESSION_LONGOPS

Многие действия в базе данных могут выполняться достаточно долго. К таким действиям относятся, в частности, восстановление с помощью Recovery Manager, сортировка

или загрузка больших объемов данных,выполнение сложных запросов, требующих распараллеливания. При выполнении этих продолжительных действий информация о ходе работы записывается в представление динамической производительности **V$SESSION_LONGOPS**, что позволяет оценить объем сделанного. Это представление можно использовать и в приложениях. В представлении отражается состояние действий, выполняющихся в базе данных более шести секунд. Другими словами, в функции сервера Oracle, которые, по предположению разработчиков, обычно выполняются дольше шести секунд, включены вызовы процедуры, вставляющей данные в представление **V$SESSION_LONGOPS**. Это не означает, что при выполнении действия продолжительностью более шести секунд в это представление автоматически будет что-то записываться. К таким действиям сейчас относятся многие функции резервного копирования и восстановления, сбора статистической информации и выполнение запросов. В каждой новой версии Oracle появляются новые действия.

Изменения в этом представлении сразу же доступны для других сеансов, т.е. транзакцию фиксировать не нужно. Можно контролировать ход процесса, изменяющего это представление, из другого сеанса с помощью запросов к представлению **V$SESSION_LONGOPS**. Разработчики могут добавлять строки в это представление. Обычно приложение вставляет и изменяет одну строку, но при необходимости можно вставлять и несколько.

Процедура для установки значений в этом представлении имеет следующие параметры:

```
PROCEDURE SET_SESSION_LONGOPS
 Argument Name                  Type                    In/Out Default?

 RINDEX                         BINARY_INTEGER          IN/OUT
 SLNO                           BINARY_INTEGER          IN/OUT
 OP_NAME                        VARCHAR2                IN     DEFAULT
 TARGET                         BINARY_INTEGER          IN     DEFAULT
 CONTEXT                        BINARY_INTEGER          IN     DEFAULT
 SOFAR                          NUMBER                  IN     DEFAULT
 TOTALWORK                      NUMBER                  IN     DEFAULT
 TARGET_DESC                    VARCHAR2                IN     DEFAULT
 UNITS                          VARCHAR2                IN     DEFAULT
```

Эти параметры описаны ниже.

- **RINDEX**. Указывает серверу, какую строку в представлении **V$SESSION_LONGOPS** необходимо изменить. Если в качестве значения этого параметра указать **DBMS_APPICATION_INFO.SET_SESSION_LONGOPS_NOHINT**, в это представление будет автоматически вставлена новая строка, индекс которой будет записан в **RINDEX**. При указании в последующих вызовах процедуры **SET_SESSION_LONGOPS** этого индекса в качестве значения параметра **RINDEX** будет изменяться добавленная строка.

- **SLNO**. Служебное значение. Первоначально надо передать значение **NULL**, а полученное в результате выполнения значение — игнорировать. При каждом вызове надо передавать одно и то же значение.

- **OP_NAME**. Имя продолжительно выполняющегося процесса. Его длина не должна превышать 64 байт, а в качестве значения необходимо задавать строку, по которой легко определить, что именно выполняется.
- **TARGET**. Обычно используется для передачи идентификатора объекта, с которым выполняется продолжительное действие (например, идентификатор таблицы, в которую загружаются данные). Можно передать любое значение, в том числе NULL.
- **CONTEXT**. Число, задаваемое пользователем. Это число должно быть информативным для пользователя. Передать можно любое число.
- **SOFAR**. В качестве значения можно передавать любое число, но если это число будет представлять собой процент или другую количественную характеристику уже выполненной части действия, сервер сможет автоматически оценить, сколько времени осталось до завершения действия. Например, если необходимо обработать 25 объектов и на обработку каждого из них уходит примерно одинаковое время, можно задать в качестве значения параметра **SOFAR** количество обработанных объектов, а общее их количество передать как следующий параметр, **TOTALWORK**. Сервер определит, сколько времени потребовалось для выполнения уже сделанной части, и оценит время, необходимое для завершения действия.
- **TOTALWORK**. В качестве значения можно передавать любое число, но имеет смысл сопоставлять его со значением параметра **SOFAR**. Если **SOFAR** представляет собой процент от **TOTALWORK**, отражающий ход выполнения действия, сервер сможет вычислить, сколько времени осталось до завершения действия.
- **TARGET_DESC**. Этот параметр описывает значение TARGET, представленное выше. Если в качестве параметра TARGET передан идентификатор объекта, параметр может содержать имя объекта с этим идентификатором.
- **UNITS**. Описательный параметр, задающий единицу измерения для значений параметров **SOFAR** и **TOTALWORK**. Это могут быть, например, файлы, итерации или вызовы.

Перечисленные выше значения может устанавливать пользователь. Если обратиться к представлению **V$SESSION_LONGOPS**, то в нем можно обнаружить намного больше столбцов, чем описано выше:

```
ops$tkyte@ORA8I.WORLD> desc v$session_longops
 Name                                      Null?    Type

 SID                                                NUMBER
 SERIAL#                                            NUMBER
 OPNAME                                             VARCHAR2(64)  **
 TARGET                                             VARCHAR2(64)  **
 TARGET_DESC                                        VARCHAR2(32)  **
 SOFAR                                              NUMBER **
 TOTALWORK                                          NUMBER **
 UNITS                                              VARCHAR2(32)  **
 START_TIME                                         DATE
 LAST_UPDATE_TIME                                   DATE
 TIME_REMAINING                                     NUMBER
```

```
ELAPSED_SECONDS                 NUMBER
CONTEXT                         NUMBER **
MESSAGE                         VARCHAR2(512)
USERNAME                        VARCHAR2(30)
SQL_ADDRESS                     RAW(4)
SQL_HASH_VALUE                  NUMBER
QCSID                           NUMBER
```

Значения столбцов, помеченных двумя звездочками (**), может устанавливать пользователь.

Остальные столбцы имеют следующие значения.

- Столбцы **SID** и **SERIAL#** используются при соединении с представлением **V$SESSION** для получения информации о сеансе.
- Столбец **START_TIME** содержит время создания записи (обычно это время первого вызова процедуры **DBMS_APPLICATION_INFO.SET_SESSION_LONGOPS,** с помощью которого и была создана строка).
- Столбец **LAST_UPDATE_TIME** представляет время последнего вызова процедуры **SET_SESSION_LONGOPS.**
- Столбец **TIME_REMAINING** содержит предполагаемое время, оставшееся до завершения действия. Его значение вычисляется как:

  ROUND(ELAPSED_SECONDS*((TOTALWORK/SOFAR)-1)).
- Столбец **ELAPSED_SECONDS** содержит количество секунд, прошедших с начала выполнения продолжительного действия до последнего изменения строки.
- Столбец **MESSAGE** — производный. Он представляет собой конкатенацию значений столбцов **OPNAME, TARGET_DESC, TARGET, SOFAR, TOTALWORK** и **UNITS,** описывающую выполняемое действие.
- Значение в столбце **USERNAME** — имя пользователя, выполняющего действие.
- Значения столбцов **SQL_ADDRESS** и **SQL_HASH_VALUE** можно использовать для поиска в представлении **V$SQLAREA** последнего SQL-оператора, выполненного процессом.
- Значение столбца **QCSID** используется при распараллеливании запроса. Это идентификатор сеанса-координатора параллельного запроса.

Итак, что же можно получить с помощью данного представления? Небольшой пример поможет прояснить это. Выполним в одном сеансе следующий блок кода:

```
tkyte@TKYTE816> declare
  2     l_nohint number default
          dbms_application_info.set_session_longops_nohint;
  3     l_rindex number default l_nohint;
  4      l_slno   number;
  5  begin
  6     for i in 1 .. 25
  7      loop
  8          dbms_lock.sleep(2);
  9        dbms_application_info.set_session_longops
```

```
 10          (rindex =>  l_rindex,
 11           slno   =>  l_slno,
 12  op_name  => 'my long running operation',
 13           target  =>  1234,
 14           target_desc => '1234 is my target',
 15           context     => 0,
 16           sofar    =>    i,
 17           totalwork   => 25,
 18           units       => 'loops'
 19          );
 20      end loop;
 21  end;
 22  /
```

Этот блок кода представляет собой продолжительное действие, выполняющееся в течение 50 секунд (вызов **DBMS_LOCK.SLEEP** просто приостанавливает выполнение на две секунды). В другом сеансе можно следить за ходом данного сеанса с помощью представленного ниже запроса (описание использованной в нем утилиты **PRINT_TABLE** см. в главе 23):

```
tkyte@TKYTE816> begin
  2     print_table('select b.*
  3                    from v$session a, v$session_longops b
  4                   where a.sid = b.sid
  5                     and a.serial# = b.serial#');
  6  end;
  7  /
SID                           : 11
SERIAL»                       : 635
OPNAME                        : my long running operation
TARGET                        : 1234
TARGET_DESC                   : 1234 is my target
SOFAR                         : 2
TOTALWORK                     : 25
UNITS                         : loops
START_TIME                    : 28-apr-2001 16:02:46
LAST_UPDATE_TIME              : 28-apr-2001 16:02:46
TIME_REMAINING                : 0
ELAPSED_SECONDS               : 0
CONTEXT                       : 0
MESSAGE                       : my long running operation: 1234 is my
                                target 1234: 2 out of 25 loops done
USERNAME                      : TKYTE
SQL_ADDRESS                   : 036C3758
SQL_HASH_VALUE                : 1723303299
QCSID                         : 0

PL/SQL procedure successfully completed.

ops$tkyteeORA8I.WORLD> /
SID                           : 11
SERLAL#                       : 635
```

```
OPNAME                         : my long running operation
TARGET                         : 1234
TARGET_DESC                    : 1234 is my target
SOFAR                          : 6
TOTALWORK                      : 25
UNITS                          : loops
START_TIME                     : 28-apr-2001 16:02:46
LAST_UPDATE_TIME               : 28-apr-2001 16:02:55
TIME_REMAINING                 : 29
ELAPSED_SECONDS                : 9
CONTEXT                        : 0
MESSAGE                        : my long running operation: 1234 is my
                                 target 1234: 6 out of 25 loops done
USERNAME                       : TKYTE
SQL_ADDRESS                    : 036C3758
SQL_HASH_VALUE                 : 1723303299
QCSID                          : 0

PL/SQL procedure successfully completed.

ops$tkyte@ORA8I.WORLD> /
SID                            : 11
SERIAL#                        : 635
OPNAME                         : my long running operation
TARGET                         : 1234
TARGET_DESC                    : 1234 is my target
SOFAR                          : 10
TOTALWORK                      : 25
UNITS                          : loops
START_TIME                     : 28-apr-2001 16:02:46
LAST_UPDATE_TIME               : 28-apr-2001 16:03:04
TIME_REMAINING                 : 27
ELAPSED_SECONDS                : 18
CONTEXT                        : 0
MESSAGE                        : my long running operation: 1234 is my
                                 target 1234: 10 out of 25 loops done
USERNAME                       : TKYTE
SQL_ADDRESS                    : 036C3758
SQL_HASH_VALUE                 : 1723303299
QCSID                          : 0

PL/SQL procedure successfully completed.
```

Может возникнуть вопрос: зачем представление **V$SESSION_LONGOPS** соединяется с представлением **V$SESSION**, если из **V$SESSION** не выбирается информация? Дело в том, что представление **V$SESSION_LONGOPS** содержит строки как для текущего, так и для прежних сеансов. По завершении сеанса это представление не очищается. Данные остаются, пока какой-нибудь другой сеанс не использует повторно соответствующий слот. Поэтому, чтобы получить информацию только для текущих сеансов, необходимо использовать соединение или подзапрос.

Этот достаточно простой пример демонстрирует, что из представления **V$SESSION_LONGOPS** можно получить информацию, весьма ценную для пользователей и администраторов базы данных, которым необходимо контролировать ход выполнения продолжительных хранимых процедур, пакетных заданий, отчетов и т.п. Добавив лишь несколько вызовов, можно получить точную информацию в производственной среде. Вместо того чтобы гадать, что именно происходит в задании и сколько оно будет выполняться, можно получить точную информацию о том, что сделано, и обоснованную оценку времени, необходимого для завершения работы.

Резюме

В этом разделе мы рассмотрели пакет **DBMS_APPLICATION_INFO**, о котором мало кто знает и использует его. Пакет можно и нужно использовать в любом приложении для регистрации в базе данных, что позволит администратору базы данных или пользователю, отвечающему за сервер, определить, какие приложения с ним работают. Очень важно использовать представление **V$SESSION_LONGOPS** в приложениях, выполняющихся дольше нескольких секунд. Только это позволит показать, что процесс не "висит", а выполняет необходимые действия. Oracle Enterprise Manager (OEM) и многие инструментальные средства независимых производителей обращаются к этому представлению и автоматически отображают информацию, которую приложения в нем устанавливают.

Пакет DBMS_JAVA

Пакет **DBMS_JAVA** весьма загадочен. Это PL/SQL-пакет, но он не описан в справочном руководстве *Supplied PL/SQL Packages Reference.* Пакет создавался для поддержки Java на сервере, так что можно предположить, что он описан в справочном руководстве *Supplied Java Packages Reference* (но там о нем тоже ничего нет). Описан этот пакет в руководстве *Oracle8i Java Developer's Guide.* Я уже многократно упоминал и использовал его в примерах, не вникая в детали. Пришло время рассмотреть процедуры этого пакета и описать их назначение и особенности использования.

В пакет **DBMS_JAVA** входит почти 60 процедур и функций, но лишь некоторые из них действительно полезны для разработчиков ПО. Основная часть пакета предназначена для отладчиков (точнее, для тех, кто создает программы-отладчики). Кроме того, пакет включает ряд служебных подпрограмм и утилит экспорта/импорта. Все эти функции и процедуры мы рассматривать не будем.

Функции LONGNAME и SHORTNAME

Это служебные функции для преобразования коротких, 30-символьных идентификаторов (все идентификаторы Oracle — не длиннее 30 символов) в длинные имена Java, и наоборот. В словаре данных обычно находятся хешированные имена Java-классов, загруженных в базу данных. Причина в том, что их исходные имена — слишком длинные, а сервер такие имена не поддерживает. Эти две функции позволяют получить реальное имя по короткому (значению столбца **OBJECT_NAME** в представлении **USER_OBJECTS),** а также короткое имя для полного имени. Вот пример использования этих функций пользователем **SYS** (которому принадлежит много фрагментов Java-кода, если в базе данных установлена поддержка Java):

```
sys@TKYTE816> column long_nm format a30 word_wrapped
sys@TKYTE816> column short_nm format a30

sys@TKYTE816> select dbms_java.longname(object_name) long_run,
  2       dbms_java.shortname(dbms_java.longname(object_name)) short_nm
  3    from user_objects where object_type = 'JAVA CLASS'
  4     and rownum < 11
  5  /

LONG_NM                        SHORT_NM


com/visigenic/vbroker/ir/Const/1001a851_ConstantDefImpl
antDefImpl

oracle/sqlj/runtime/OraCustomD/10076b23_OraCustomDatumClosur
atumClosure

com/visigenic/vbroker/intercep/10322588_HandlerRegistryHelpe
tor/HandlerRegistryHelper


10 rows selected.
```

Как видите, применив функцию LONGNAME к значению OBJECT NAME, можно получить исходное имя Java-класса. Если применить к этому длинному имени функцию SHORTNAME, мы снова получим короткое хешированное имя, используемое внутренне сервером Oracle.

Установка опций компилятора

Для компилятора Java в базе данных можно задавать большинство опций, причем двумя способами. Опции можно задавать в командной строке при использовании процедуры **loadjava** или в таблице **JAVA$OPTIONS.** Опция, установленная в командной строке, всегда используется вместо соответствующего значения из таблицы **JAVA$OPTIONS.** Это, конечно же, происходит только при использовании компилятора Java в базе данных Oracle. Если используется отдельный компилятор Java вне базы данных (например, в среде JDeveloper), опции надо устанавливать в соответствующей среде.

Можно устанавливать три опции компилятора, каждая из которых связана с прекомпилятором SQLJ (это Java-прекомпилятор, преобразующий встроенные операторы SQL в вызовы интерфейса JDBC), встроенным в базы данных. Эти опции представлены в следующей таблице.

Опция	*Назначение*	*Возможные значения*
ONLINE	Выполнять ли проверку типов при компиляции (online) или при выполнении	True/False
DEBUG	Компилировать ли Java-код с включенной отладкой. Аналог вызова **javac -g** из командной строки	True/False
ENCODING	Кодировка исходного файла для компилятора	Стандартно используется кодировка **Latin1**

Стандартные значения опций выделены полужирным.

Я продемонстрирую использование средств пакета **DBMS_JAVA** для установки опций компилятора на примере опции **online** прекомпилятора **SQLJ**. Обычно эта опция имеет стандартное значение True, что требует от прекомпилятора SQLJ проверять семантику SQLJ-кода. Это означает, что прекомпилятор **SQLJ** обычно проверяет существование всех упомянутых в коде объектов базы данных, соответствие типов хост-переменных и т.п. Если необходимо делать эти проверки при выполнении (например, потому, что таблицы, на которые ссылается SQLJ-код, еще не созданы, но хотелось бы загрузить код в базу без ошибок), можно отключить проверку семантики с помощью процедуры **DBMS_JAVA.SET_COMPILER_OPTION**.

Для иллюстрации используем следующий фрагмент кода. В нем выполняется попытка вставить данные в таблицу, которой в базе данных еще нет:

```
tkyte@TKYTE816> create or replace and compile
  2  java source named "bad_code"
  3   as
  4  import java.sql.SQLException;
  5
  6  public class bad_code extends Object
  7   {
  8  public static void wont_work() throws SQLException
  9   {
 10     #sql {
 11         insert into non_existent_table values (1)
 12      };
 13   }
 14   }
 15   /

Java created.

tkyte@TKYTE816> show errors java source "bad_code"
Errors for JAVA SOURCE bad_code:

LINE/COL ERROR

0/0     bad_code:7: Warning: Database issued an error: PLS-00201:
          identifier 'NON_EXISTENT_TABLE' must be declared
0/0      insert into non_existent_table values ( 1 )
0/0                   ^^^^^^^^^^^^^^^^^^^^
0/0
0/0       #sql {
0/0
0/0       Info: 1 warnings
```

Теперь установим опции компилятора **ONLINE** значение **FALSE.** Для этого необходимо отключиться от сервера и подключиться снова. Проблема связана с тем, что среда времени выполнения языка Java проверяет существование таблицы **JAVA$OPTIONS** только при запуске. Если таблицы нет, повторных попыток прочитать из нее данные в сеансе не делается. Процедура **DBMS_JAVA.SET_COMPILER_OPTION** создаст эту таблицу автоматически, но только если она вызвана до запуска среды времени выполнения языка Java. Так что сеанс придется начинать заново.

В следующем примере мы организуем новый сеанс и убедимся, что таблицы **JAVA$OPTIONS** нет. Мы установим опцию компилятора и увидим, что таблица автоматически создана. Наконец, мы создадим такую же Java-функцию, как в примере выше, и увидим, что теперь она компилируется без предупреждений благодаря установленной опции компилятора:

```
tkyte@TKYTE816> disconnect
Disconnected from Oracle8i Enterprise Edition Release 8.1.6.0.0 —
-> Production
With the Partitioning option
JServer Release 8.1.6.0.0 - Production

tkyte@TKYTE816> connect tkyte/tkyte
Connected.
tkyte@TKYTE816> column value format a10
tkyte@TKYTE816> column what format a10

tkyte@TKYTE816> select * from java$options;
select * from java$options
              *
ERROR at line 1:
ORA-00942: table or view does not exist
tkyte@TKYTE816> begin
  2          dbms_java.set_compiler_option
  3          (what       => 'bad_code',
  4           optionName => 'online',
  5           value      => 'false') ;
  6  end;
  7  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select * from java$options;

WHAT       OPT                  VALUE

bad_code   online               false

tkyte@TKYTE816> create or replace and compile
  2  java source named "bad_code"
  3  as
  4  import java.sql.SQLException;
  5
  6  public class bad_code extends Object
  7  {
  8  public static void wont_work() throws SQLException
  9  {
 10     #sql {
 11          insert into non_existent_table values (1)
 12     };
 13  }
 14  }
 15  /

Java created.
```

```
tkyte@TKYTE816> show errors java source "bad_code"
No errors.
```

В данном случае процедура **SET_COMPILER_OPTION** вызывается с тремя параметрами.

- **WHAT**. Шаблон, с которым надо сопоставлять код. Обычно Java-программы используют пакеты, поэтому полное имя будет иметь вид **a.b.c.bad_code**, а не просто bad_code. Если необходимо установить опцию для пакета **a.b.c**, это можно сделать. В результате для любого кода, имя которого соответствует шаблону **a.b.c**, будет использоваться соответствующая опция, если только нет более точной спецификации, тоже соответствующей данному пакету. Если опции заданы для значений WHAT, равных **a.b.c** и **a.b.c.bad_code**, будет использоваться опция для **a.b.c.bad_code**, поскольку она соответствует большей части имени.
- **OPTIONNAME**. Одно из трех значений: **ONLINE, DEBUG** или **ENCODING**.
- **VALUE**. Значение соответствующей опции.

С процедурой **SET_COMPILER_OPTION** связаны еще две подпрограммы.

- **GET_COMPILER_OPTION**. Эта функция возвращает значение указанной опции компилятора, даже если оно не отличается от стандартного.
- **RESET_COMPILER_OPTION**. Эта процедура удаляет из таблицы **JAVA$OPTIONS** строки, соответствующие шаблону **WHAT** и имеющие указанное значение в столбце **OPTIONNAME**.

Вот примеры использования обеих подпрограмм. Начнем с использования **GET_COMPILER_OPTION**, чтобы узнать значение опции **online**:

```
tkyte@TKYTE816> set serveroutput on
tkyte@TKYTE816> begin
  2         dbms_output.put_line
  3         (dbms_java.get_compiler_option(what       => 'bad_code',
  4                                        optionName => 'online'));
  5  end;
  6  /
false

PL/SQL procedure successfully completed.
```

А затем сбросим ее с помощью процедуры **RESET_COMPILER_OPTION**:

```
tkyte@TKYTE816> begin
  2         dbms_java.reset_compiler_option(what       =>  'bad_code',
  3                       optionName                   => 'online');
  4  end;
  5  /

PL/SQL procedure successfully completed.
```

Теперь убедимся, что функция **GET_COMPILER_OPTION** всегда возвращает значение указанной опции компилятора, даже если таблица **JAVA$OPTIONS** пуста (при вызове **RESET_COMPILER_OPTION** соответствующая строка была удалена):

```
tkyte@TKYTE816> begin
  2           dbms_output.put_line
  3           (dbms_java.get_compiler_option(what       => 'bad_code',
  4                                          optionName => 'online'));
  5  end;
  6  /
true

PL/SQL procedure successfully completed.

tkyte@TKYTE816>select * from java$options;

no rows selected
```

Процедура SET_OUTPUT

Эта процедура аналогична команде **SET SERVEROUTPUT ON** в SQL*Plus. Точно так же, как выполнение этой команды включает вывод строк с помощью пакета **DBMS_OUTPUT,** вызов **DBMS_JAVA.SET_OUTPUT** обеспечивает вывод результатов функций **System.out.println** и **System.err.print** на экран в SQL*Plus. Если не выполнить вызовы:

```
SQL> set serveroutput on size 1000000
SQL>  exec  dbms_java.set_output(1000000)
```

перед выполнением хранимой процедуры на языке Java из среды SQL*Plus, то все сообщения, выдаваемые с помощью вызовов **System.out.println,** будут записываться в трассировочный файл в каталоге на сервере, задаваемом параметром инициализации **USER_DUMP_DEST.** Эта процедура используется при отладке хранимых процедур на языке Java, поскольку позволяет помещать в код выдачу отладочной информации в виде вызовов **System.out.println** во многом аналогично вызовам **DBMS_OUTPUT.PUT_LINE** в PL/SQL-коде. В дальнейшем можно отключить выдачу отладочной информации в Java-коде, перенаправив ее в "корзину".

Так что, если вас интересовало, куда делись результаты вызовов **System.out** в хранимой процедуре на языке Java, вы теперь знаете ответ. Они выдаются в трассировочный файл. Теперь вы сможете перенаправить их на экран в среде SQL*Plus.

Процедуры loadjava и dropjava

Эти процедуры обеспечивают интерфейс для языка PL/SQL, позволяющий выполнять функции утилит командной строки **loadjava** и **dropjava.** Как и следовало ожидать для служебных процедур, при их вызове не надо указывать опцию **-u имя_пользователя/пароль** или задавать тип используемого JDBC-драйвера — вы ведь уже подключились к базе данных. Процедуры загрузят Java-объекты в текущую схему. Эти процедуры имеют следующие прототипы:

```
PROCEDURE loadjava(options varchar2)
PROCEDURE loadjava(options varchar2, resolver varchar2)
PROCEDURE dropjava(options varchar2)
```

Их можно использовать для загрузки файла **activation8i.zip,** который используется также в разделе приложения А, посвященном пакету UTL_SMTP. Более детальную информацию об интерфейсе JavaMail можно найти на странице **http://java.sun.com/products/javamail/index.html.** Рассмотрим пример:

```
sys@TKYTE816> exec dbms_java.loadjava('-r -v -f -noverify -synonym -g
-> public c:\temp\activation8i.zip')
initialization complete
loading : com/sun/activation/registries/LineTokenizer
creating : com/sun/activation/registries/LineTokenizer
loading : com/sun/activation/registries/MailcapEntry
creating : com/sun/activation/registries/MailcapEntry
loading : com/sun/activation/registries/MailcapFile
creating : com/sun/activation/registries/MailcapFile
loading : com/sun/activation/registries/MailcapParseException
creating : com/sun/activation/registries/MailcapParseException
```

Процедуры управления правами

Это весьма странные процедуры. Выполните команду **DESCRIBE** для пакета **DBMS_JAVA** в базе данных и поищите в результатах упоминание о процедуре **GRANT_PERMISSION. Вы** его не найдете, хотя точно известно, что такая процедура должна быть (я несколько раз демонстрировал ее использование). Она существует, как и ряд других функций, связанных с правами доступа. Я опишу использование подпрограмм **GRANT_PERMISSION/REVOKE_PERMISSION.** Подробное описание использования процедур управления правами и всех соответствующих опций можно найти в руководстве *Oracle Java Developers Guide.* Процедуры управления правами описаны в главе 5, *Security for Oracle 8i Java Applications,* этого руководства.

В Oracle 8.1.5 точность установки привилегий для Java была очень низкой. Можно было указать только **JAVAUSERPRIV** или **JAVASYSPRIV.** Это напоминает ситуацию, когда в базе данных имеются только роли **RESOURCE** и **DBA** — обе они предоставляют пользователям слишком много возможностей. В версии Oracle 8.1.6 реализация Java в базе данных поддерживает классы защиты Java 2. Теперь имеется очень детальный набор привилегий, которые можно избирательно предоставлять и отбирать, аналогично набору привилегий в базе данных. Описание и обсуждение соответствующих классов привилегий можно найти на Web-странице **http://java.sun.eom/j2se/l.3/docs/api/java/security/Permission.html.**

Итак, для установки привилегий будем использовать две процедуры — **GRANT_PERMISSION** и **REVOKE_PERMISSION.** Вопрос в том, как определить необходимые привилегии? Проще всего установить Java-код, выполнить его и узнать, чего ему не хватает для нормальной работы. Например, обратимся к разделу, посвященному пакету **UTL_SMTP. В** нем я создаю хранимую процедуру **SEND** для посылки сообщения по электронной почте. Я также демонстрирую там, какие две привилегии необходимо предоставить с помощью процедуры **GRANT_PERMISSION,** чтобы процедура **SEND** заработала. Необходимые привилегии я определил именно так — выполняю SEND и читаю сообщения об ошибках. Например:

```
tkyte@TKYTE816> set serveroutput on size 1000000
tkyte@TKYTE816> exec dbms_java.set_output(1000000)
PL/SQL procedure successfully completed.
tkyte@TKYTE816> declare
  2    ret_code number;
  3  begin
  4     ret_code := send(
  5                    p_from => 'me@here.com',
  6                    p_to => 'me@here.com',
  7                    p_cc => NULL,
  8                    p_bcc => NULL,
  9                    p_subject => 'Use the attached Zip file',
 10                    p_body => 'to send email with attachments....',
 11                    p_smtp_host=> 'aria.us.oracle.com',
 12                    p_attachment_data => null,
 13                    p_attachment_type => null,
 14                    p_attachment_file_name => null);
 15     if ret_code = 1 then
 16       dbms_output.put_line ('Сообщение послано успешно...');
 17     else
 18       dbms_output.put_line ('Послать сообщение не удалось...');
 19    end if;
 20 end;
 21 /
java.security.AccessControlException: the Permission (java.util.Property
Permission * read,write) has
not been granted by dbms_java.grant_permission to
SchemaProtectionDomain(TKYTE|PolicyTableProxy(TKYTE))
```

Все предельно ясно. В сообщениях говорится, что пользователю **TKYTE** необходима привилегия **java.util.PropertyPermission** с именем * и параметрами **read** и **write**. Вот откуда я узнал, что надо выполнить:

```
sys@TKYTE816> begin
  2    dbms_java.grant_permission(
  3       grantee => 'TKYTE',
  4       permission_type => 'java.util.PropertyPermission',
  5       permission_name => '*',
  6       permission_action => 'read,write'
  7   );
```

После этого при попытке выполнения я получил следующее сообщение об ошибке:

```
java.security.AccessControlException: the Permission (java.net.SocketPer
mission aria.us.oracle.com resolve) has not been granted by
dbms_java.grant_permission to
SchemaProtectionDomain(TKYTE|PolicyTableProxy(TKYTE))
```

Предоставив соответствующую привилегию, я узнал, что кроме **RESOLVE** необходима привилегия CONNECT. Вот почему я выполнил:

```
  8    dbms_java.grant_permission(
  9       grantee => 'TKYTE',
 10       permission_type => 'java.net.SocketPermission',
```

```
11        permission_name => '*',
12        permission_action => 'connect,resolve'
13      );
14    end;
15    /
```

И получил все необходимые соответствующей схеме привилегии. Обратите внимание, что в качестве значения **permission_name** я задал *, чтобы процедура могла разрешать имя любого хоста и подключаться к любому хосту, а не только к моему серверу SMTP.

Процедура **REVOKE_PERMISSION** выполняет действие, обратное **GRANT_PERMISSION** Она работает именно так, как ожидалось. Если передать те же параметры, что были переданы процедуре **GRANT_PERMISSION,** она отберет соответствующую привилегию у текущей схемы.

Резюме

В этом разделе мы рассмотрели использование средств пакета **DBMS_JAVA** для выполнения различных действий, необходимых для поддержки Java-кода. Мы начали с описания того, как сервер Oracle, ограничивающий длину имен 30 символами, обрабатывает очень длинные имена, используемые в языке Java. Для каждого длинного имени Java-сервер создает уникальный 30-символьный идентификатор с помощью хеширования. Пакет **DBMS_JAVA** предоставляет функцию преобразования короткого имени в соответствующее ему длинное имя и функцию преобразования длинного имени в короткое с помощью хеширования.

Затем было рассмотрено использование средств пакета **DBMS_JAVA** для установки, получения и сброса опций компилятора Java. Мы разобрались, как таблица **JAVA$OPTIONS** используется для постоянного хранения стандартных опций компилятора и как вернуть этим опциям стандартные значения. Затем мы кратко рассмотрели процедуру **SET_OUTPUT.** Она перенаправляет вывод результатов вызовов **System.out.println** в Java-коде в сеанс SQL*Plus или **SVRMGRL,** аналогично тому, как команда **SET SERVEROUTPUT** ON обеспечивает выдачу результатов вызовов процедур PL/SQL-пакета **DBMS_OUTPUT.** Мы также рассмотрели альтернативный способ загрузки (с помощью вызова хранимой процедуры) исходного кода на языке Java, файлов классов и Java-архивов, ставший возможным благодаря пакету **DBMS_JAVA** в версиях Oracle8i Release 2 (8.1.6) и выше. Наконец, мы изучили процедуры управления правами доступа, предоставляемые этим пакетом в версиях, начиная с Oracle8i Release 2. Эти процедуры позволяют гибко управлять привилегиями для Java-кода, устанавливая, что они могут и чего не могут делать.

Если вы используйте язык Java в базе данных Oracle, то постоянно будете применять эти процедуры при программировании.

Пакет DBMS_JOB

Пакет **DBMS_JOB** позволяет запланировать однократное или регулярное выполнение заданий в базе данных. Задание представляет собой хранимую процедуру, анонимный блок PL/SQL или внешнюю процедуру на языке C или Java. Эти задания выполняются серверными процессами в фоновом режиме. Задания можно выполнять регулярно (в 2 часа ночи каждые сутки) или однократно (выполнить задание сразу после фиксации транзакции и удалить его из очереди). Если вы знакомы с утилитами **cron** или **at** в ОС UNIX или Windows, то особенности использования пакета **DBMS_JOB** вам уже понятны. Задания выполняются в той же среде (пользователь, набор символов и т.п.), из которой они посланы на выполнение (но роли не действуют). Задания выполняются аналогично процедурам с правами создателя, т.е. роли для них не учитываются. Это можно продемонстрировать на следующем примере (процедуры, использованные в этом примере, позже будут рассмотрены подробно):

```
tkyte@TKYTE816> create table t (msg varchar2(20), cnt int);

Table created.

tkyte@TKYTE816> insert into t select 'from SQL*PLUS', count(*) from
                session_roles;

1 row created.

tkyte@TKYTE816> variable n number
tkyte@TKYTE816> exec dbms_job.submit(:n,'insert into t select ''from job'',
                count(*) from session_roles;');

PL/SQL procedure successfully completed.

tkyte@TKYTE816> print n
```

```
         N

        81

tkyte@TKYTE816> exec dbms_job.run(:n);
PL/SQL procedure successfully completed
tkyte@TKYTE816> select * from t;
MSG                            CNT

from SQL*PLUS                   10
from job                         0
```

Как видите, в среде SQL*Plus имеется 10 действующих ролей, а в среде выполнения задания — ни одной. Поскольку пользователи в качестве задания обычно вызывают хранимую процедуру, это не имеет значения, потому что при выполнении хранимой процедуры роли все равно не учитываются. Проблема возникает только при попытке выполнить процедуру, доступную через **роль.** Такая процедура не сработает, поскольку в заданиях роли не действуют.

Часто пользователи спрашивают, как лучше всего скрыть имя пользователя/пароль, связанные с пакетным заданием (например, для периодического анализа таблиц), посылаемым с помощью утилиты **cron** или ее аналогов в среде Windows NT/2000. Их беспокоит, что пароль хранится в файле (и это правильно) или явно представлен в результатах выполнения команды ps в среде UNIX и т.п. Я всегда рекомендую вообще не использовать средства ОС для планирования заданий в базе данных, а вместо этого написать хранимую процедуру, выполняющую необходимые действия, и запланировать ее выполнение с помощью средств пакета **DBMS_JOB.** При этом ни имя пользователя, ни пароль нигде не хранится, и задание выполнится только в том случае, когда доступна база данных. Если сервер базы данных не будет работать в соответствующий момент, задание, естественно, не выполнится, поскольку именно сервер базы данных и отвечает за выполнение задания.

Часто спрашивают также: как ускорить выполнение? Необходимо выполнить продолжительное действие, а пользователь не хочет ждать. Причем иногда ускорить выполнение действия нельзя. Например, я уже многие годы посылаю сообщения электронной почты с сервера базы данных. Я использовал для этого различные механизмы: каналы базы данных, пакет **UTL_HTTP,** внешние процедуры и средства языка Java. Все они работали примерно с одинаковой скоростью, но всегда — медленно. Иногда завершения работы по протоколу SMTP приходится ждать достаточно долго. Слишком долго в случае моего приложения, для которого ожидания более четверти секунды вообще **недопустимы.** Посылка сообщения по протоколу SMTP может иногда потребовать 2-3 секунды. Ускорить этот процесс нельзя, но можно создать **видимость** его более быстрого выполнения. Вместо отправки сообщения при нажатии пользователем соответствующей кнопки в приложении должно посыпаться на выполнение задание, которое будет отправлять сообщение сразу после фиксации транзакции. При этом возникает два побочных эффекта. Во-первых, действие выполняется как бы быстрее, а во-вторых, отправка сообщения становится частью транзакции. Одно из свойств пакета **DBMS_JOB** состоит в том, что задание попадает в очередь только после фиксации транзакции. При откате транзакции задание удаляется из очереди и не выполняется. С помощью пакета

DBMS_JOB мы не только создаем видимость более быстрой работы приложения, но и делаем его более надежным. Больше не будет посылаться уведомление по электронной почте из триггера при изменении строки, если это изменение затем было отменено. Либо строка изменяется, и отправляется сообщение; либо строка не изменяется, и сообщение не отправляется.

Так что пакет **DBMS_JOB** имеет широкую сферу применения. Он может включить выполнение действий, выходящих "за рамки" транзакций (таких как отправка сообщений по электронной почте или создание таблицы при вставке строки в другую таблицу) в транзакции. Он позволяет создать видимость более быстрого выполнения действий, особенно, если продолжительные действия не должны выдавать пользователю результатов. Пакет позволяет планировать и автоматизировать многие задачи, для решения которых обычно создавались сценарии вне базы данных. Это, безусловно, очень полезный пакет.

Для корректной работы пакета **DBMS_JOB** необходимо выполнить небольшую настройку сервера. Надо установить два параметра инициализации.

- **job_queue_interval.** Задает периодичность (в секундах) проверки очередей и поиска заданий, готовых к выполнению. Если задание должно выполняться раз в 30 секунд, но параметр **job_queue_interval** имеет (стандартное) значение 60, это задание не будет выполняться раз в 30 секунд — в лучшем случае, раз в 60 секунд.
- **job_queue_processes.** Задает количество фоновых процессов для выполнения заданий. Значением может быть целое число от 0 (по умолчанию) до 36. Это значение можно менять без перезапуска сервера с помощью оператора **ALTER SYSTEM SET JOB_QUEUE_PROCESSES=<nn>.** Если оставить стандартное значение 0, задания из очереди автоматически никогда выполняться не будут. Процессы обработки очередей заданий можно увидеть в среде ОС UNIX — они получают имена **ora_snpN_$ORACLE_SID,** где **N** — число (0, **1,** 2, **...,job_queue_processes-l). В** среде Windows очереди заданий обрабатываются потоками операционной системы, увидеть которые с помощью стандартных средств нельзя.

Многие системы работают со значением параметра **job_queue_interval,** равным 60 (другими словами, проверяют очереди раз в минуту), и значением параметра **job_queue_processes,** равным 1 (выполняют одновременно не более одного задания). При интенсивном использовании заданий или возможностей, для реализации которых используются задания (в частности, репликация и поддержка материализованных представлений используют очереди заданий), может потребоваться добавление дополнительных процессов и увеличение значения параметра инициализации **job_queue_processes.**

После настройки и автоматического запуска очередей заданий можно начинать их использование. Основная процедура пакета **DBMS_JOB** — процедура **SUBMIT.** Она имеет следующий интерфейс:

```
PROCEDURE SUBMIT
Argument Name                  Type                    In/Out Default?

JOB                            BINARY_INTEGER          OUT
WHAT                           VARCHAR2                IN
NEXT_DATE                      DATE                    IN     DEFAULT
INTERVAL                       VARCHAR2                IN     DEFAULT
```

```
NO_PARSE                    BOOLEAN              IN          DEFAULT
INSTANCE                    BINARY_INTEGER       IN          DEFAULT
FORCE                       BOOLEAN              IN          DEFAULT
```

Назначение аргументов процедуры **SUBMIT** описано ниже.

- **JOB.** Идентификатор задания. Присваивается **системой** (этот параметр передается в режиме **OUT).** Его можно использовать для получения информации о задании из представлений **USER_JOBS** или **DBA_JOBS** по идентификатору задания. Кроме того, некоторые процедуры, в частности **RUN** и **REMOVE,** требуют единственного параметра — идентификатора задания — для определения того, какое именно задание выполнять или удалять из очереди.
- **WHAT.** Действие, которое необходимо выполнить. Можно передавать PL/SQL-оператор или блок кода. Например, чтобы выполнить хранимую процедуру P, можно передать процедуре строку P; (включая точку с запятой). Значение параметра **WHAT** будет помещено в следующий PL/SQL-блок:

```
DECLARE
   job BINARY_INTEGER := :job;
   next_date DATE := :mydate;
   broken BOOLEAN := FALSE;
BEGIN
        WHAT
    :mydate := next_date;
    IF broken THEN :b := 1; ELSE :b := 0; END IF;
END;
```

Вот почему любой оператор надо завершать точкой с запятой (;). Чтобы **WHAT** можно было заменить соответствующим кодом, точка с запятой необходима.

- **NEXT_DATE**. Время следующего (а поскольку мы только посылаем задание — первого) выполнения задания. Стандартное значение — **SYSDATE** — означает "выполнять немедленно" (после фиксации транзакции).
- **INTERVAL.** Строка, содержащая функцию, вычисляющую время следующего выполнения задания. Можно считать, что значение этой функции выбирается с помощью оператора **'select … from dual'.** Если передать строку **sysdate+l,** сервер выполнит **SELECT sysdate+l INTO :NEXT_DATE FROM DUAL.** Ниже описаны особенности задания интервала выполнения задания, предотвращающие его смещение.
- **NO_PARSE.** Указывает, анализировался ли параметр **WHAT** при отправке задания. Анализируя строку, можно определить, выполнимо ли вообще задание. В общем случае параметру **NO_PARSE** надо всегда оставлять стандартное значение, **False.** При установке значения **True** параметр **WHAT** принимается "как есть", без проверки допустимости.
- **INSTANCE.** Этот параметр имеет значение только в режиме Parallel Server, когда сервер Oracle работает на кластере слабо связанных машин. Он задает экземпляр, на котором будет выполняться задание. По умолчанию он имеет значение **ANY_INSTANCE.**

- **FORCE.** Этот параметр также имеет значение только в режиме Parallel Server. При установке значения **True** (принятого по умолчанию) можно посылать задание с любым идентификатором экземпляра, даже если при отправке соответствующий экземпляр недоступен. При установке значения **False** отправка задания завершится неудачно, если указанный экземпляр недоступен.

В пакете **DBMS_JOB** есть и другие подпрограммы. Задание посылается на выполнение с помощью процедуры **SUBMIT,** а остальные подпрограммы позволяют манипулировать заданиями в очередях и выполнять действия по их запуску **(RUN),** удалению из очереди **(REMOVE)** и изменению (**CHANGE**). Ниже описаны наиболее часто используемые подпрограммы, включая входные данные и назначение.

Подпрограмма	*Входные данные*	*Описание*
REMOVE	номер задания	Удаляет задание из очереди. Учтите, что если задание выполняется, удаление не остановит его выполнение. Удаление из очереди гарантирует, что задание не будет выполнено снова, но уже выполняющееся задание при этом не останавливается. Для остановки выполняющегося задания необходимо прекращать работу соответствующего сеанса с помощью оператора **ALTER SYSTEM.**
CHANGE	номер задания **WHAT, NEXT_DATE, INTERVAL, INSTANCE, FORCE**	Эта процедура работает как оператор **UPDATE** для представления **JOBS.** Она позволяет изменить любой параметр задания.
BROKEN	номер задания **BROKEN** (булево значение) **NEXT_DATE**	Позволяет "разрушить" или "восстановить" задание. Разрушенное задание не выполняется. Задание, не выполнившееся успешно 16 раз подряд, автоматически помечается как разрушенное, и сервер Oracle больше не будет его выполнять.
RUN	номер задания	Выполняет задание немедленно, в приоритетном режиме (в пользовательском сеансе). Полезно при отладке не срабатывающих заданий.

Теперь, когда вы достаточно хорошо представляете устройство пакета DBMS_JOB и его возможности, рассмотрим, как выполнить задание однократно, как организовать его периодическое выполнение, как контролировать выполнение заданий и определять возникающие при этом ошибки.

Однократное выполнение задания

Многие из заданий в моей базе данных — однократные. Я часто использую пакет **DBMS_JOB** как аналог запуска процесса в фоновом режиме с помощью & в ОС UNIX или команды **start** в среде Windows. Представленный ранее пример с отправкой сооб-

щения по электронной почте относится как раз к этой категории. Я использую пакет **DBMS_JOB** не только для включения отправки сообщения в транзакцию, но и для того, чтобы ускорить выполнение этого действия. Вот одна из возможных реализаций этого действия, демонстрирующая однократное выполнение задания. Начну с небольшой хранимой процедуры для отправки сообщений по электронной почте с использованием средств стандартного пакета **UTL_SMTP**:

```
tkyte@TKYTE816> create or replace
  2  PROCEDURE send_mail (p_sender    IN VARCHAR2,
  3                       p_recipient IN VARCHAR2,
  4                       p_message   IN VARCHAR2)
  5  as
  6     — Учтите, что надо указать хост,
  7     — поддерживающий протокол SMTP и доступный для вас.
  8     — К указанному хосту вы не получите доступа, поэтому его надо
-> изменить
  9     l_mailhost VARCHAR2(255) := 'aria.us.oracle.com';
 10     l_mail_conn utl_smtp.connection;
 11  BEGIN
 12     l_mail_conn :=utl_smtp.open_connection(l_mailhost, 25);
 13     utl_smtp.helo(l_mail_conn, l_mailhost);
 14     utl_smtp.mail(l_mail_conn, p_sender);
 15     utl_smtp.rcpt(l_mail_conn, p_recipient);
 16     utl_smtp.open_data(l_mail_conn) ;
 17     utl_smtp.write_data(l_mail_conn, p_message);
 18     utl_smtp.close_data(l_mail_conn);
 19     utl_smtp.quit(l_mail_conn) ;
 20  end;
 21  /

Procedure created.
```

Теперь, чтобы определить продолжительность работы, я выполню эту процедуру дважды:

```
tkyte@TKYTE816> set serveroutput on
tkyte@TKYTE816> declare
  2          l_start number := dbms_utility.get_time;
  3  begin
  4          send_mail('anyone@outthere.com',
  5            'anyone@outthere.com',            'Привет!');
  6          dbms_output.put_line
  7      (round((dbms_utility.get_time-l_start)/100,      2)  ||
  8             ' seconds');
  9  end;
 10  /
.81 seconds

PL/SQL procedure successfully completed.

tkyte@TKYTE816> /
.79 seconds

PL/SQL procedure successfully completed.
```

Похоже, она выполняется примерно восемь десятых секунды, в лучшем случае. Для меня это слишком долго. Можно ускорить выполнение, точнее, создать видимость этого. Я использую задания для видимости ускорения работы и получаю при этом преимущества отправки сообщений в рамках транзакции.

Начнем с создания таблицы для хранения сообщений и процедуры, которая сможет посылать находящиеся в ней сообщения. Эта процедура и будет выполняться как фоновое задание. Вопрос в том, зачем для хранения сообщений используется таблица? Почему просто не передать текст сообщения как параметр задания? Причина в использовании связываемых переменных и разделяемого пула. Поскольку все задания будут создаваться с параметром WHAT, а сервер будет просто выполнять эту строку, надо помнить, что значение параметра WHAT окажется в разделяемом пуле. Можно посылать задания и так:

```
dbms_job.submit(x, 'send_mail(''someone@there.com'',
                                ''someone0there.com'', ''Привет!'');');
```

но в результате в разделяемом пуле окажутся сотни тысяч уникальных операторов, что отрицательно скажется на производительности сервера. Поскольку предполагается рассылка большого количества сообщений (больше одного — уже много, и использовать связываемые переменные при этом обязательно), надо иметь возможность посылать задания вида:

```
dbms_job.submit(x,    'background_send_mail(константа);');
```

Оказывается, этого очень легко добиться. Достаточно создать таблицу с полями для каждого передаваемого параметра при отправке задания на выполнение (в данном случае, отправитель, адресат и само сообщение) и первичным ключом. Например:

```
tkyte@TKYTE816>  create table send_mail_data(id          number primary key,
  2                                  sender    varchar2(255),
  3                                  recipient varchar2(255),
  4                                  message   varchar2(4000),
  5                                  senton    date default NULL);

Table created.
```

Я добавил в таблицу первичный ключ — столбец **ID**, и дату отправки сообщения в столбце **senton**. Мы будем использовать эту таблицу не только для организации очереди исходящих сообщений, но и как постоянный журнал, в котором будут регистрироваться все отправленные сообщения (пригодится, поверьте мне, когда кто-то скажет: "А меня не предупреждали..."). Осталось только придумать, как генерировать значение ключа для таблицы и передавать его фоновому процессу в виде строковой константы. К счастью, пакет **DBMS_JOB** содержит все необходимое для решения этой проблемы. При отправке задания на выполнение пакет автоматически создает для него уникальный идентификатор и возвращает его вызывающему. Поскольку блок кода, в который помещается переданное значение параметра **WHAT**, содержит идентификатор задания, мы можем его передавать. Процедура **FAST_SEND_MAIL** будет выглядеть так:

```
tkyte@TKYTE816>  create or replace
  2  PROCEDURE fast_send_mail (p_sender    IN VARCHAR2,
  3                            p_recipient IN VARCHAR2,
```

```
  4                              p_message    IN VARCHAR2)
  5  as
  6           l_job   number;
  7  begin
  8          dbms_job.submit(l_job,  'background_send_mail(JOB);');
  9           insert into send_mail_data
 10          (id, sender, recipient, message)
 11           values
 12          (l_job, p_sender, p_recipient, p_message);
 13  end;
 14  /

Procedure created.
```

Эта процедура будет посылать на выполнение задание **BACKGROUND_SEND_MAIL** и передавать ему параметр **JOB.** Если обратиться к описанию параметра **WHAT,** вы увидите, что соответствующий блок кода включает три локальных переменных, к которым можно обращаться, — мы и передаем процедуре одну из них. Сразу после этого мы вставляем сообщение в таблицу очереди для последующей отправки. Итак, пакет **DBMS_JOB** создает первичный ключ, а затем мы вставляем этот первичный ключ и соответствующие данные в таблицу. Вот и все. Теперь необходимо создать несложную процедуру **BACKGROUND_SEND_MAIL:**

```
tkyte@TKYTE816> create or replace
  2  procedure background_send_mail(p_job in number)
  3  as
  4           l_rec   send_mail_data%rowtype;
  5  begin
  6           select * into l_rec
  7             from send_mail_data
  8            where id = p_job;
  9
 10          send_mail(l_rec.sender, l_rec.recipient, l_rec.message);
 11          update send_mail_data set senton = sysdate where id = p_job;
 12  end;
 13  /

Procedure created.
```

Она читает сохраненные данные, вызывает медленно работающую процедуру **SEND_MAIL,** а затем изменяет соответствующую запись, отражая в ней факт отправки сообщения по электронной почте. Теперь можно выполнить процедуру FAST_SEND_MAIL и определить, насколько быстро она выполняется:

```
tkyte@TKYTE816> declare
  2          l_start number := dbms_utility.get_time;
  3  begin
  4           fast_send_mai1('panda@panda.com',
  5               'snake@snake.com',              'Привет!');
  6           dbms_output.put_line
  7          (round((dbms_utility.get_time-l_start)/100, 2)  ||
  8            ' seconds');
  9  end;
```

```
 10  /
.03 seconds

PL/SQL procedure successfully completed.

tkyte@TKYTE816> /
.02 seconds

PL/SQL procedure successfully completed.
```

С точки зрения пользователя процедура **FAST_SEND_MAIL** работает **в 26-40** раз **быстрее,** чем исходная. На самом же деле она работает не быстрее, но создает видимость быстрого выполнения (именно это и имеет значение). Фактическая отправка сообщения будет выполнена в фоновом режиме после фиксации транзакции. Об этом важно помнить. При выполнении этого примера не забудьте выполнить **COMMIT,** иначе сообщение никогда не будет послано. Задание не будет доступно в очереди для соответствующих процессов, пока транзакция не будет зафиксирована (в сеансе можно будет увидеть задание в представлении **USER_JOBS,** но процессы обработки очередей его не увидят, пока не будет зафиксирована транзакция). Не считайте это ограничением. На самом деле это полезное свойство, с помощью которого мы только что включили отправку сообщений по электронной почте в транзакцию. Если откатить транзакцию, сообщение не будет послано. После фиксации транзакции оно будет отправлено.

Текущие задания

Еще одно стандартное применение пакета **DBMS_JOB** — для организации периодического выполнения заданий в базе данных. Как уже упоминалось, многие пользователи пытаются применять для выполнения заданий в базе данных утилиты ОС **cron** или **at,** но сталкиваются при этом с проблемами защиты пароля и т.п. Я всегда предлагаю использовать очереди заданий. Они не только избавляют от необходимости хранения регистрационной информации, но и гарантируют выполнение заданий только в случае работоспособности и доступности сервера базы данных. В случае сбоя сервер будет повторно пытаться выполнить задание. Например, если при первой попытке выполнения задания удаленная база данных недоступна, задание возвращается в очередь и делается ,попытка выполнить его снова. Сервер делает это 16 раз, с каждым разом немного увеличивая время ожидания, прежде чем пометит задание как "разрушенное". Подробнее об этом мы поговорим в подразделе "Контроль заданий и поиск ошибок". Утилиты **cron** и **at** автоматически этого не сделают. Кроме того, поскольку задания выполняются в базе данных, с помощью запросов можно определить их статус: когда последний раз выполнялось задание и выполнялось ли вообще, и т.п. Вся информация находится в одном месте.

Другие средства сервера Oracle, такие как репликация и материализованные представления, неявно используют очереди заданий при реализации своих функциональных возможностей. Изменения моментальных снимков и обновления материализованных представлений выполняются с помощью заданий, вызывающих соответствующие хранимые процедуры.

Предположим, необходимо ежесуточно в 3 часа ночи выполнять анализ всех таблиц в определенной схеме. Для этого можно использовать следующую хранимую процедуру:

```
scott@TKYTE816> create or replace procedure analyze_my_tables
  2  as
  3  begin
  4     for x in (select table_name from user_tables)
  5      loop
  6         execute immediate
  7           'analyze table ' || x.table_name || ' compute statistics';
  8     end loop;
  9  end;
 10  /

Procedure created.
```

Чтобы запланировать ее выполнение **сегодня ночью** в 3 часа (точнее, завтра утром), а затем ежесуточно в 3 часа ночи, можно использовать следующий вызов:

```
scott@TKYTE816>declare
  2     l_job number;
  3  begin
  4     dbms_job.submit(job       => l_job,
  5                     what      => 'analyze_my_tables;',
  6                     next_date => trunc(sysdate)+1+3/24,
  7                     interval  => 'trunc(sysdate)+1+3/24');
  8  end;
  9  /

PL/SQL procedure successfully completed.

scott@TKYTE816>select job, to_char(sysdate,'dd-mon'),
  2                    to_char(next_date,'dd-mon-yyyy hh24:mi:ss'),
  3                   interval, what
  4  from user_jobs
  5  /

 JOB TO_CHA TO_CHAR(NEXT_DATE,'D INTERVAL                WHAT

  33 09-jan 10-jan-2001 03:00:00 trunc(sysdate)+1+3/24 analyze_my_tables;
```

Итак, в следующий раз это задание выполнится в 3 часа ночи 10 января. Для этого мы передали реальную дату, а не строку, как для параметра **interval.** Мы использовали функций для работы с датами, так что при выполнении, независимо от времени вызова, всегда будет возвращаться 3 часа **следующего** утра. Это важно. Точно такую же функцию, но в виде строки мы передали в качестве значения параметра **INTERVAL.** Используется функция, всегда возвращающая 3 утра завтрашнего дня, независимо от времени ее выполнения. Это предотвращает *смещение заданий* (jobs sliding). Может показаться, что, поскольку первый раз задание выполняется в 3 часа утра, можно задать интервал **sysdate+1.** Если выполнить это вычисление в 3 утра во вторник, в результате мы получим 3 утра среды. Получим, **если** задание гарантированно выполнится в указанное время, но это вовсе не обязательно. Задания в очереди обрабатываются последовательно, в соответствии с указанным временем выполнения. При наличии одного процесса обработки очереди сообщений и двух заданий, назначенных на 3 утра, очевидно, что одно из них не выполнится точно в 3 утра. Придется подождать, пока завершится выполнение первого задания. Даже при отсутствии заданий, назначенных на то же вре-

мя, очереди заданий просматриваются периодически, например каждые 60 секунд. Задание, назначенное на выполнение в 3 утра, может быть выбрано из очереди в 3:00:45 утра. Если использовать функцию **sysdate+l,** в следующий раз задание может быть поставлено на выполнение в 3:00:46 утра. На следующий день в 3:00:45 утра задание еще не будет готово для выполнения и выполнится при следующем просмотре очереди, в 3:01:45 утра. Время выполнения задания медленно сдвигается. Однако это не самое худшее. Предположим, на следующий день в 3 утра с таблицами будут работать и проанализировать их не удастся. Хранимая процедура не сработает и будет возвращена в очередь для выполнения. Теперь это задание окажется смещенным на несколько минут позже 3 утра. Поэтому, чтобы предотвратить смещение времени выполнения заданий, необходимо использовать функцию, возвращающую **фиксированный** момент времени, если выполнение в **конкретный** момент времени является существенным. Если важно, чтобы задание выполнялось именно в 3 утра, надо использовать функцию, **всегда** возвращающую время 3 утра, независимо от времени ее выполнения.

Многие из таких "не смещающихся" функций реализовать очень легко, другие — намного сложнее. Например, однажды меня попросили создать задание, собирающее статистическую информацию с помощью **STATSPACK** с понедельника по пятницу ровно в 7 часов утра и 3 часа дня. Значение параметра **INTERVAL** для этого задания задать непросто, но давайте рассмотрим для начала псевдокод:

```
if время - до 15:00
then
   вернуть 15:00 СЕГОДНЯ
   (другими словами, если мы выполняем это в 7 утра, надо выполнить
   задание сегодня в 3 часа дня)
else
   вернуть 7 утра через 3 дня (если сегодня пятница) либо 1 день (в
   противном случае)
end if
```

Осталось реализовать эту логику в виде симпатичной функции **DECODE** или, если для вас это слишком сложно, в виде PL/SQL-функции. Я использовал интервал:

```
decode(sign(15-to_char(sysdate,'hh24')),
            1, trunc(sysdate)+15/24,
            trunc(sysdate+decode(to_char(sysdate,'d'), 6, 3, 1))+7/24)
```

Функция **decode** начинается с вычисления значения **SIGN(15-TO_CHAR(SYSDATE,'HH24')). SIGN** — это функция, возвращающая **-1**, **0** или **1**, если переданное ей выражение имеет, соответственно, отрицательное, нулевое и положительное значение. Если значение было положительным, предполагается, что вызов произошел до 3 часов дня (до 15 часов), поэтому в следующий раз надо будет выполнить задание в **TRUNC(SYSDATE)+15/24** (сегодня в 15 часов). С другой стороны, если **sign** возвращает 0 или -1, надо вернуть значение **TRUNC(SYSDATE + DECODE(TO_CHAR(SYSDATE,'D'), 6, 3, l))+7/24.** Здесь с помощью функции **DECODE** мы определяем день недели, чтобы узнать, сколько добавлять дней — три (в пятницу, чтобы вернуть понедельник) или один (в остальные рабочие дни). Полученное количество дней мы добавляем к значению **SYSDATE,** усекаем время до полуночи и добавляем 7 часов.

Бывает, что смещение даты вполне допустимо и даже желательно. Например, если необходимо собирать статистическую информацию из представлений V$ каждые 30 минут в процессе работы сервера, вполне допустимо использовать интервал **SYSDATE+1/24/2,** добавляющий к текущему моменту времени полчаса.

Нетривиальное планирование

Бывает, как в рассмотренном выше примере, что значение **NEXT_DATE** вычислить одним оператором SQL сложно или время следующего выполнения задания определяется сложным алгоритмом. В этом случае можно определить время следующего выполнения в самом задании.

В начале раздела было сказано, что задание выполняется в следующем PL/SQL-блоке:

```
DECLARE
   job BINARY_INTEGER := :job;
   next_date DATE := :mydate;
   broken BOOLEAN := FALSE;
BEGIN
         WHAT
    :mydate := next_date;
    IF broken THEN :b := 1; ELSE :b := 0; END IF;
END;
```

Вы уже видели (в подразделе "Однократное выполнение задания"), как использовать доступное в блоке значение **JOB.** Его можно использовать как первичный ключ для таблицы параметров, чтобы в максимальной степени обеспечить совместное использование SQL-операторов сеансами. Можно воспользоваться и значением переменной **NEXT_DATE.** Как демонстрирует представленный выше блок кода, сервер Oracle использует для установки значения переменной **NEXT_DATE** связываемую переменную **:mydate** в качестве **входного** параметра процедуры, но он также получает ее значение после выполнения **WHAT** (заданной процедуры). Если процедура изменит значение переменной **NEXT_DATE,** сервер Oracle будет использовать его в качестве времени следующего выполнения задания. Для иллюстрации этого приема создадим небольшую процедуру P, которая будет выдавать сообщение в таблицу T и устанавливать значение **NEXT_DATE:**

```
tkyte@TKYTE816>create table t (msg varchar2(80));
Table created.

tkyte@TKYTE816>create or replace
  2 procedure p(p_job in number, p_next_date in OUT date)
  3  as
  4     l_next_date date default p_next_date;
  5  begin
  6     p_next_date := trunc(sysdate)+1+3/24;
  7
  8      insert into t values
  9      ('Next date имела значение "' ||
 10       to_char(l_next_date,'dd-mon-yyyyhh24:mi:ss') ||
```

```
11        '" Next date ИМЕЕТ значение ' ||
12         to_char(p_next_date,'dd-mon-yyyy hh24:mi:ss'));
13  end;
14  /

Procedure created.
```

Теперь пошлем эту процедуру на выполнение, используя метод, рассмотренный в подразделе "Однократное выполнение задания". Другими словами, не зададим значение параметра INTERVAL:

```
tkyte@TKYTE816> variable n number

tkyte@TKYTE816> exec dbms_job.submit(:n, 'p(JOB,NEXT_DATE);');
PL/SQL procedure successfully completed.

tkyte@TKYTE816> select what, interval,
  2         to_char(last_date,'dd-mon-yyyyhh24:mi:ss') last_date,
  3         to_char(next_date,'dd-mon-yyyyhh24:mi:ss') next_date
  4    from user_jobs
  5   where job = :n
  6  /

WHAT                          INTERVAL LAST_DATE            NEXT_DATE

p(JOB,NEXT_DATE);             null                          28-apr-2001
18:23:01
```

В данном случае процедуре переданы только параметры **JOB** и **NEXT_DATE**. Их значения будут получены при выполнении. Как видите, это задание еще не выполнялось (столбец **LAST_DATE** имеет значение Null), а параметр **INTERVAL** получил пустое значение, так что значение **NEXT_DATE** будет вычисляться как **SELECT NULL FROM DUAL**. Обычно это означает, что задание выполнится один раз, после чего будет удалено из очереди заданий. Однако при выполнении этого задания оказывается:

```
tkyte@TKYTE816> exec dbms_job.run(:n);

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select * from t;

MSG

Next date имела значение "" Next date ИМЕЕТ значение 29-apr-2001 03:00:00

tkyte@TKYTE816> select what, interval,
  2     to_char(last_date,'dd-non-yyyyhh24:mi:ss')     last_date,
  3         to_char(next_date,'dd-mon-yyyyhh24:mi:ss') next_date
  4    from user_jobs
  5   where job = :n
  6  /

WHAT                  INTERVAL  LAST_DATE             NEXT_DATE

p(JOB,NEXT_DATE);     null      28-apr-2001 18:23:01 29-apr-2001 03:00:00
```

что параметр NEXT_DATE получает другое значение. Это значение NEXT_DATE было вычислено в самой процедуре, и задание снова оказалось в очереди. Пока задание бу-

дет устанавливать непустое значение **NEXT_DATE,** оно будет оставаться в очереди. Если когда-нибудь после успешного выполнения не будет установлено значение **NEXT_DATE,** задание будет удалено из очереди.

Этот прием пригодится для заданий, в которых значение **NEXT_DATE** сложно вычислить или оно зависит от данных в других таблицах.

Контроль заданий и обнаружение ошибок

Для контроля заданий в базе данных используется три основных представления.

- **USER_JOBS.** Список всех заданий, посланных текущим зарегистрированным пользователем. У этого представления есть также общедоступный синоним, **ALL_JOBS. ALL_JOBS** содержит ту же информацию, что и **USER_JOBS.**
- **DBA_JOBS.** Полный список всех заданий, находящихся в очередях базы данных.
- **DBA_JOBS_RUNNING.** Список выполняющихся заданий.

Обычно представление **USER_JOBS** доступно всем пользователям, а представления **DBA_*** — только пользователям с привилегией **DBA** или получившим привилегию **SELECT** непосредственно для этих представлений. В этих представлениях находится следующая информация.

- **LAST_DATE/LAST_SEC**. Когда последний раз выполнялось задание. **LAST_DATE** - столбец типа **DATE. LAST_SEC** — строка символов, содержащая только время (в формате **часов:минут:секунд).**
- **THIS_DATE/HIS_SEC.** Если задание в настоящий момент выполняется, в этом столбце будет время начала выполнения. Как и пара столбцов **LAST_DATE/ LAST_SEC,** столбец **THIS_DATE** содержит дату и время, а столбец **THIS_SEC** - символьную строку, в которой указано только время.
- **NEXT_DATE/NEXT_SEC.** Время, когда задание будет выполнено в следующий раз.
- **TOTAL_TIME.** Общее время выполнения задания в секундах. Включает время выполнения предыдущих прогонов — это суммарное значение.
- **BROKEN.** Флаг **Yes/No,** показывающий, что задание разрушено. Разрушенные задания не выполняются процессами обработчик очередей. Задание разрушается после 16 неудачных попыток выполнения. Для разрушения задания можно использовать процедуру **DBMS_JOB.BROKEN** (что временно предотвращает его выполнение).
- **INTERVAL.** Функция, возвращающая дату, которая вызывается в начале следующего выполнения задания, чтобы определить, когда снова выполнять задание.
- **FAILURES.** Сколько раз подряд задание не было успешно выполнено. При успешном выполнении задания в этом столбце устанавливается значение 0.
- **WHAT.** Текст задания.
- **NLS_ENV.** Среда NLS (National Language Support — поддержка национальных языков), в которой будет выполняться задание. Включает язык, формат даты, фор-

мат чисел и т.п. Среда NLS полностью наследуется из среды, откуда было послано задание. При изменении этой среды и повторной отправке задания оно будет выполняться в измененной среде.

- INSTANCE. Имеет смысл только в режиме Parallel Server. Это идентификатор экземпляра, на котором может выполняться задание, а в представлении **DBA_JOBS_RUNNING** — экземпляра, на котором оно выполняется.

Предположим, в этих представлениях обнаружено задание с положительным значением в столбце **FAILURES**. Где найти сообщения об ошибках для этого задания? Они не хранятся в базе данных, а записываются в *журнал уведомлений* (alert log) базы данных. Например, создана следующая процедура:

```
tkyte@TKYTE816>create or replace procedure run_by_jobs
  2  as
  3          l_cnt   number;
  4  begin
  5         select user_id into l_cnt from all_users;
  6          — другой необходимый код
  7  end;
  8  /

Procedurecreated.

tkyte@TKYTE816>variable n number
tkyte@TKYTE816> exec dbms_job.submit(:n, 'run_by_jobs;');

PL/SQL procedure successfully completed.

tkyte@TKYTE816> commit;

Commit complete.

tkyte@TKYTE816> exec dbms_lock.sleep(60);

PL/SQL procedure successfully completed.

tkyte@TKYTE816>select job, what, failures
  2    from user_jobs
  3   where job = :n;
      JOB WHAT                                  FAILURES

       35 run_by_jobs;                                 1
```

Если в базе данных больше одного пользователя (во всех базах данных их больше), эта процедура, определенно, **не сработает.** Оператор **SELECT ... INTO** всегда будет возвращать слишком много строк; **при** программировании была допущена ошибка. Однако, поскольку она происходит в фоновом режиме, причину ошибки трудно определить. К счастью, сообщение об ошибке записывается в журнал уведомлений базы данных. Если открыть этот файл в редакторе, в конце файла можно будет найти следующее:

```
Tue Jan 09 13:07:51 2001
Errors in file C:\oracle\admin\tkyte816\bdump\tkyte816SNP0.TRC:

ORA-12012: error on auto execute of job 35
ORA-01422: exact fetch returns more than requested number of rows
```

```
ORA-06512: at "SCOTT.RUN_BY_JOBS", line 5
ORA-06512: at line 1
```

Сообщение свидетельствует о том, что задание 35 (наше задание) выполнить не удалось. Что важнее, в сообщении указана **причина** неудачи. Такие же сообщения об ошибках выдаются при попытке выполнения процедуры в среде SQL*Plus. Эта информация принципиально важна для определения причины сбоя задания. На ее основе можно, изменив задание, обеспечить его корректную работу.

Вот, пожалуй, и все о контроле выполнения заданий. Надо либо следить за журналом уведомлений, **alert.log** (это администратор базы данных должен делать всегда), либо периодически просматривать представление **DBA_JOBS,** чтобы убедиться в успешном выполнении заданий.

Резюме

Пакет DBMS_JOB — замечательное средство сервера, обеспечивающее выполнение действий в фоновом режиме. Его можно использовать для автоматизации рутинных операций, таких как анализ таблиц, архивирование и очистка временных таблиц, да и любых других. Его можно использовать в приложениях для обеспечения сравнительно быстрого выполнения продолжительных действий (причем, выполнение это будет действительно быстрым с точки зрения пользователя). Пакет позволяет не писать для каждой ОС специфические сценарии, периодически выполняющие действия в базе данных. Более того, он избавляет от необходимости "зашивать" имена пользователей и пароли в сценарии для регистрации на сервере. Задание всегда выполняется от имени пользователя, который его послал, — регистрироваться вообще не нужно. Наконец, в отличие от средств периодического выполнения заданий операционных систем, эти задания **сервера базы данных** выполняются только в случае доступности сервера. Если система не работала в момент запланированного выполнения задания, оно не будет выполнено (очевидно, если сервер не работает, то и очереди заданий не просматриваются). Пакет **DBMS_JOB** — надежное средство, для которого я нашел множество применений.

Пакет DBMS_LOB

DBMS_LOB — это стандартный пакет для работы с большими объектами (Large **OB**jects — **LOB**s) в базе данных. Большими объектами называют данные новых типов, появившиеся начиная с версии Oracle 8. Большие объекты поддерживают хранение и обработку до 4 Гбайт данных в одном столбце. Они заменяют считающиеся теперь ненужными типы данных **LONG** и **LONG RAW**. Использование типов данных **LONG** в Oracle было связано с множеством ограничений, в том числе:

- в таблице можно было иметь только один столбец типа **LONG** или **LONG RAW**;
- если объем данных превышал 32 Кбайт, с ними нельзя было работать в хранимых процедурах;
- их нельзя было изменять по частям;
- многие операции в базе данных, например **INSERT INTO T SELECT LONG_COL FROM T2**, для столбцов типа **LONG** и **LONG RAW** не поддерживались;
- их нельзя было указывать в конструкции **WHERE**;
- таблицы со столбцами типа **LONG** и **LONG RAW** нельзя было реплицировать;
- и т.д...

Типы данных LOB не имеют всех этих ограничений.

Вместо писания всех функций и процедур пакета **DBMS_LOB** (их около 25) я собираюсь ответить на наиболее часто задаваемые вопросы об использовании пакета **DBMS_LOB** и больших объектов. Подпрограммы пакета либо чрезвычайно просты, либо хорошо описаны в документации Oracle. При использовании больших объектов основными являются два руководства:

- *Oracle8i Supplied PL/SQL Packages Reference.* В этом руководстве можно найти обзор пакета **DBMS_LOB** и описание каждой его подпрограммы, включая все входные и выходные параметры. Соответствующий раздел пригодится как справочник. Его надо бегло просмотреть, чтобы знать основные возможности обработки больших объектов.
- *Oracle8i Application Developer's Guide — Large Objects (LOBs).* Руководство, полностью посвященное описанию программирования с использованием больших объектов в различных средах и языках. Каждый разработчик, предполагающий использовать большие объекты, должен его прочитать.

Кроме того, многое в работе с большими объектами зависит от языка программирования. Способ выполнения определенных действий в языке Java отличается от принятого в языке C, при программировании на PL/SQL и т.д. Вот почему корпорация Oracle разработала отдельные руководства *Application Developer's Guide* для языков PL/SQL, Pro*C, COBOL, VB и Java, а также библиотеки OCI, посвященные взаимодействию с большими объектами в каждом из языков. Кроме того, есть еще упомянутое исчерпывающее руководство *Application Developer's Guide,* посвященное большим объектам и полезное, независимо от используемого языка. Я рекомендую всем, кто собирается использовать большие объекты в приложениях, прочитать это руководство, а также специфическое руководство для выбранного языка разработки. В этих руководствах можно найти ответы на большинство вопросов.

В этом разделе я собираюсь ответить на часто задаваемые вопросы о больших объектах, начиная с "как показать их на Web-странице?" и заканчивая "как преобразовать данные типа BLOB в тип CLOB, и наоборот? " — об этом недостаточно хорошо сказано в стандартной документации. С большими объектами работать очень просто, если разобраться в пакете **DBMS_LOB** (описан в руководстве *Oracle 8i Supplied PL/SQL Packages Reference),* и если вы этого еще не сделали, ознакомьтесь с его описанием сейчас, прежде чем читать данный раздел. Я предполагаю, что вы готовы к работе с большими объектами.

Как загружать большие объекты?

Методов загрузки больших объектов немного. В главе 9, например, я демонстрировал загрузку больших объектов в базу данных с помощью средств **SQLLDR.** Кроме того, в предлагаемом корпорацией Oracle руководстве *Application Developer's Guide* для каждого языка продемонстрировано, как создавать и получать значения больших объектов на этом языке (у всех есть небольшие отличия). Я, однако, думаю, что при наличии каталога с файлами для загрузки проще всего использовать тип данных **BFILE,** объект **DIRECTORY** и процедуру **LOADFROMFILE.**

В главе 9 (в первой части книги — *прим. научн. ред.),* посвященной загрузке данных, мы подробно рассмотрели использование процедуры **DBMS_LOB.LOADFROMFILE.** Детальную информацию вы сможете найти в этой главе. Кроме того, в представленном далее подразделе "Преобразования" рассматривается полный пример загрузки данных типа **CLOB** с помощью процедуры **LOADFROMFILE.**

Функция substr

Небольшое примечание относительно функции **substr,** предлагаемой в пакете **DBMS_LOB.** Все остальные реализации функции **substr,** с которыми мне приходилось сталкиваться (в том числе в языках SQL и PL/SQL), принимают следующие аргументы, в указанном порядке:

```
substr(строка, с_какого_символа, сколько_символов);
```

Так что **substr('hello', 3, 2)** даст в результате строку || — третий и четвертый символы (начиная с символа 3 выбрать 2 символа). В функции **DBMS_LOB.SUBSTR,** однако, порядок параметров другой:

```
dbms_lob.substr(большой_объект, сколько_символов, с_какого_символа)
```

Поэтому аналогичный вызов функции **substr** из пакета **DBMS_LOB** вернет строку **ell.** Небольшой тестовый пример подтверждает это:

```
tkyte@TKYTE816> create table t (str varchar2(10), lob clob);

Table created.

tkyte@TKYTE816> insert into t values ('hello', 'hello');

1 row created.

tkyte@TKYTE816> select substr(str, 3, 2),
  2       dbms_lob.substr(lob, 3, 2) lob
  3    from t
  4  /

SU LOB

11 ell
```

Я постоянно передаю аргументы не в том порядке. Это — одна из тех вещей, о которых надо помнить!

Оператор SELECT FOR UPDATE в языке Java

Чтобы изменить большой (не временный) объект в базе данных, строка, содержащая его, должна быть заблокирована соответствующим сеансом. Обычно это не учитывают те, кто пишет программы на языке Java/JDBC. Рассмотрим представленную далее небольшую программу на Java. Она:

- вставляет запись (поэтому резонно предположить, что эта запись заблокирована);
- читает локатор только что созданного большого объекта;
- пытается использовать этот локатор большого объекта в процедуре **DBMS_LOB.WRITEAPPEND.**

При выполнении этой Java-программы всегда выдается сообщение об ошибке:

```
java Test
java.sql.SQLException: ORA-22920: row containing the LOB value is not locked
ORA-06512: at "SYS.DBMS_LOB", line 715
ORA-06512: at line 1
```

Оказывается, вставленный большой объект больше не заблокирован сеансом. Это печальный побочный эффект стандартного режима "поддержки транзакций" протокола JDBC — по умолчанию транзакции не поддерживаются! Фиксация выполняется немедленно после каждого оператора. Если в следующем приложении не добавить вызов **conn.setAutoCommit (false);** сразу же после **getConnection,** оно не будет работать. Эта строка кода (по моему мнению) должна быть первой после **любого подключения** в программе, использующей интерфейс JDBC!

```
import java.sql.*;
import java.io.*;
import oracle.jdbc.driver.*;
import oracle.sql.*;
// Для выполнения этого приложения нужна следующая таблица:
// create table demo (id int primary key, theBlob blob);

class Test {

public static void main (String args [])
   throws SQLException , FileNotFoundException, IOException
{
    DriverManager.registerDriver
      (new oracle.jdbc.driver.OracleDriver());

    Connection conn = DriverManager.getConnection
         ("jdbc:oracle:thin:@aria:1521:ora8i",
          "scott", "tiger");

    // Если хотите, чтобы программа сработала, уберите комментарий со
    // следующей строки!
    // conn.setAutoCoamit(false);

    Statement stmt - conn.createStatement();

    // Вставляем в таблицу пустой BLOB
    // При первом вызове создадим его.
    stmt.execute
    ("insert into demo (id,theBlob)  " +
      "values (1,empty_blob())");
    // Теперь прочитаем его, чтобы можно было загрузить данные.
    ResultSet rset = stmt.executeQuery
                     ("SELECT theBlob  " +
                        "FROM demo "+
                       "where id = 1");

    if(rset.next())
    (
        // Получить BLOB для загрузки.
        BLOB l_mapBLOB = ((OracleResultSet)rset).getBLOB(1);
        // Вот данные, которые мы в него загрузим.
        File binaryFile - new File("/tmp/binary.dat");
        FileInputStream instream =
              new FileInputStream(binaryFile);

        // Мы будем загружать примерно по 32 Кбайт за раз. Это
        // максимальный фрагмент, поддерживаемый пакетом dbms_lob
        // (ограничение языка PL/SQL).
```

```
        int chunk = 32000;
        byte[] l_buffer = new byte[chunk];
        int l_nread = 0;

        // Используем простую процедуру writeappend для добавления
        // фрагмента файла в конец BLOB-бьекта.
        OracleCallableStatement cstmt =
            (OracleCallableStatement)conn.prepareCall
            ( "begin dbms_lob.writeappend( :1, :2, :3 ); end;" );
        // Читаем и записываем, пока не загрузим весь файл.
        cstmt.registerOutParameter(1, OracleTypes.BLOB);
        while ((l_nread= instream.read(l_buffer)) != -1)
        {
            cstmt.setBLOB(1, l_mapBLOB);
            cstmt.setInt(2, l_nread);
            cstmt.setBytes(3, l_buffer);
            cstmt.executeUpdate();
            l_mapBLOB=cstmt.getBLOB(1);
        }
        // Закрываем входной файл и завершаем оператор.
        instream.close();
        cstmt.close();
    )
    // Завершаем остальные операторы.
    rset.close();
    stmt.close();
    conn.close ();
}
}
```

Это общее ограничение протокола JDBC, влияющее, в частности, и на работу с большими объектами. Я не знаю, сколько разработчиков с удивлением обнаруживало, что интерфейс по умолчанию автоматически фиксирует транзакцию — это ведь должно делать приложение. Этого могут ожидать только разработчики, ранее использовавшие интерфейс ODBC! Аналогичное действие выполняет протокол ODBC в стандартном для него режиме автоматической фиксации.

Преобразования

Часто пользователи хранят данные в объектах типа **BLOB,** но иногда нужно представить их как данные типа **CLOB.** Типичный пример: в столбец типа **BLOB** загружены двоичные и текстовые данные, и необходимо проанализировать текст. Анализировать данные типа **BLOB** сложно, поскольку сервер постоянно пытается преобразовать их в шестнадцатиричное представление, что нежелательно. В других случаях имеются данные типа **LONG** или **LONG RAW,** которые хотелось бы обрабатывать как данные типов **CLOB** или **BLOB,** поскольку функциональные возможности обработки этих типов намного превосходят те, что поддерживаются для типов **LONG** и **LONG RAW.**

К счастью, эти преобразования легко выполнить. Можно преобразовать:

- данные типа **BLOB** в **VARCHAR2**;
- **VARCHAR2** - в **RAW**;
- данные типа **LONG** — в **CLOB**;
- данные типа **LONG RAW** — **BLOB**.

Рассмотрим сначала преобразование типа **BLOB** в **VARCHAR2**, и наоборот, а затем разберемся с преобразованиями типов **LONG** в **CLOB** и **LONG RAW** в **BLOB**.

Преобразование типа BLOB в VARCHAR2 и обратно

В пакет **UTL_RAW** входят две полезные подпрограммы для работы с данными типа **BLOB**. Более детально пакет **UTL_RAW** мы рассмотрим в соответствующем разделе приложения. Пока речь идет о следующих подпрограммах:

- **CAST_TO_VARCHAR2.** Принимает данные типа **RAW** и меняет тип на **VARCHAR2**. Никакого преобразования данных фактически не происходит — речь идет только об изменении типа.
- **CAST_TO_RAW**. Принимает данные типа **VARCHAR2** и меняет тип на **RAW**. И в этом случае данные не изменяются — изменяется только тип данных.

Итак, если известно, что данные типа **BLOB** содержат текстовую информацию в соответствующей кодировке, эти функции действительно полезны. Используем рассмотренную ранее программу **LOADFROMFILE** для загрузки набора файлов в столбец типа **BLOB**. Хотелось бы просматривать значения в этом столбце в среде SQL*Plus (с маскировкой любых "недопустимых" символов, являющихся причиной некорректной работы программы SQL*Plus). Для этого можно использовать средства пакета **UTL_RAW**. Сначала загрузим ряд файлов в таблицу **DEMO**:

```
scott@DEV816> create table demo
  2  (id          int primary key,
  3   theBlob     blob
  4  )
  5  /

Table created.

scott@DEV816> create or replace directory my_files as '/export/home/tkyte';

Directory created.

scott@DEV816> create sequence blob_seq;

Sequence created.

scott@DEV816> create or replace
  2 procedure load_a_file(p_dir_name in varchar2,
  3                       p_file_name in varchar2)
  4  as
  5      l_blob    blob;
```

```
 6      l_bfile   bfile;
 7  begin
 8      — Сначала необходимо создать большой объект в базе данных.
 9      — Для загрузки нужен пустой объект типа CLOB, BLOB или большой
10      — объект, созданный с помощью вызова CREATE TEMPORARY.
11
12     insert into demo values (blob_seq.nextval, empty_blob())
13     returning theBlob into l_Blob;
14
15      — Затем открываем файл (объект типа BFILE),
16      — из которого будем загружать данные.
17
18     l_bfile := bfilename(p_dir_name, p_file_name);
19     dbms_lob.fileopen(l_bfile);
20
21
22      — После этого вызываем LOADFROMFILE, загружая в только что
23      — созданный CLOB все содержимое файла (объекта типа BFILE),
24      — который только что открыли.
25     dbms_lob.loadfromfile(l_blob, l_bfile,
26                           dbms_lob.getlength(l_bfile));
27
28      — Закрываем файл (объект типа BFILE), чтобы
29      — избежать возможной нехватки дескрипторов файла.
30
31     dbms_lob.fileclose(l_bfile);
32  end;
33  /

Procedure created.

scott@DEV816> exec load_a_file('MY_FILES', 'clean.sql');

PL/SQL procedure successfully completed.

scott@DEV816> exec load_a_file('MY_FILES', 'expdat.dmp');

PL/SQL procedure successfully completed.
```

Итак, я загрузил два файла. Один из них — сценарий, над которым я сейчас работаю, **clean.sql.** Другой — файл экспорта **expdat.dmp,** подвернувшийся под руку. Теперь я собираюсь написать функцию, которую можно будет вызывать в SQL-операторах и позволяющую просматривать любой 4000-байтовый фрагмент данных типа **BLOB** в среде SQL*Plus. Просматривать можно не более 4000 байт, поскольку именно такое ограничение в SQL налагается на размер данных типа **VARCHAR2.** Представленная ниже функция **CLEAN** работает аналогично функции **SUBSTR** для обычной строки, но принимает параметр типа **BLOB** и необязательные параметры **FROM_BYTE** и **FOR_BYTES.** Это позволяет выбирать и выдавать подстроку объекта типа **BLOB.** Обратите внимание, как используется функция **UTL_RAW.CAST_TO_VARCHAR2** для преобразования типа **RAW** в тип **VARCHAR2.** Если не использовать эту функцию, байты данных типа **RAW** перед помещением в переменную типа **VARCHAR2** будут преобразовываться в шестнадцатеричное представление. С помощью этой функции мы просто меняем тип данных с **RAW** на **VARCHAR2,** не выполняя никаких преобразований:

```
scott@DEV816> create or replace
  2  function clean(p_raw in blob,
  3                 p_from_byte in number default 1,
  4                 p_for_bytes in number default 4000)
  5  return varchar2
  6  as
  7      l_tmp varchar2(8192) default
  8             utl_raw.cast_to_varchar2(
  9                 dbms_lob.substr(p_raw,p_for_bytes,p_from_byte)
 10                                  );
 11      l_char   char(1);
 12      l_return varchar2(16384);
 13      l_whitespace varchar2(25) default
 14                  chr(13) || chr(10) || chr(9);
 15      l_ws_char    varchar2(50) default
 16                   'rnt';
 17
 18  begin
 19     for i in 1 .. length(l_tmp)
 20      loop
 21         l_char := substr(l_tmp, i, 1);
 22
 23         — Если символ — "печатный" (а не управляющий), ничего с ним
 24         — делать не надо. Если это \, добавить еще один символ
 25          — \, поскольку мы будем заменять символы новой строки и
 26         — табуляции последовательностями \n, \t и т.д., поэтому
 27         — надо различать в файле текст \n и символ новой строки.
 28
 29         if (ascii(l_char) between 32 and 127)
 30          then
 31              l_return := l_return || l_char;
 32             if (l_char = '\' ) then
 33                  l_return := l_return || '\';
 34              end if;
 35
 36          — Если символ — пробельный, заменить его
 37         — специальным символом типа \r, \n, \t
 38
 39         elsif (instr(l_whitespace, l_char) > 0)
 40          then
 41              l_return := l_return ||
 42                     '\' ||
 43                    substr(l_ws_char, instr(l_whitespace,l_char), 1);
 44
 45          — Вместо всех остальных непечатных символов
 46          — просто выдать точку ('.').
 47
 48          else
 49             l_return := l_return ||  '.';
 50          end if;
 51      end loop;
 52
```

```
 53     — Теперь надо вернуть первые 4000 байт, поскольку больше
 54     — язык SQL все равно не позволит увидеть. После обработки в
 55     — строке может оказаться более 4000 символов, поскольку CHR(10)
 56     — превратится в \n (используется два байта) и т.д., т.е. это
 57      — необходимо.
 58      return substr(l_return,1,4000);
 59  end;
 60  /

Function created.

scott@DEV816> select id,
  2       dbms_lob.getlength(theBlob) len,
  3       clean(theBlob,30,40) piece,
  4       dbms_lob.substr(theBlob,40,30) raw_data
  5    from demo;

        ID   LEN PIECE                RAW_DATA

         1  3498 \ndrop sequence     0A64726F702073657175656E636520
                 blob_seq;\n\ncreate  626C6F625F7365713B0A0A63726561
                 table d              7465207461626C652064

         2  2048 TE\nRTABLES\nl024\nO 54450A525441424C45530A31303234
                 \n28\n4000\n.......  0A300A32380A343030300A0001001F
                 .   .   .   .   .    00010001000000000000
```

Как видите, теперь можно просматривать текстовые части данных типа BLOB в среде SQL*Plus, как обычный текст, воспользовавшись функцией CLEAN. Если использовать функцию **DBMS_LOB.SUBSTR,** возвращающую значение типа **RAW,** мы получим результат в шестнадцатиричном виде. Просматривая шестнадцатиричное представление, можно убедиться, что первый байт первого объекта типа **BLOB** имеет значение **0A,** или **CHR(10)** — это символ новой строки. В текстовом представлении большого объекта можно увидеть, что функция CLEAN преобразовала 0A в \n (символ новой строки). Это подтверждает, что функция выполнена, как предполагалось. Во втором объекте типа **BLOB** мы видим много двоичных нулей (значений 00 в шестнадцатиричном представлении) в обычном представлении содержимого файла **expdat.dmp.** В функции CLEAN, как видите, они преобразуются в точки, поскольку подобные специальные символы, при выдаче непосредственно на терминал, будут выдаваться в нераспознаваемом виде (как мусор).

Помимо функции **CAST_TO_VARCHAR2** пакет **UTL_RAW** содержит функцию **CAST_TO_RAW.** Как было показано ранее, в объект типа **BLOB** можно поместить обычный текст. Если для изменения этих данных надо использовать строки, пришлось бы преобразовывать их в шестнадцатиричный вид. Например, следующий оператор:

```
scott6DEV816> update demo
  2     set theBlob = 'Hello World'
  3   where id - 1
  4  /
   set theBlob = 'Hello World'
                 *
ERROR at line 2:
ORA-01465: invalid hex number
```

не работает. При неявном преобразовании данных из типа **VARCHAR2 в RAW** предполагается, что строка **Hello World** состоит из шестнадцатиричных цифр. Сервер Oracle берет первые два байта, преобразует их из шестнадцатиричного в десятичный вид и присваивает полученное значение первому байту данных типа **RAW,** и т.д. Надо либо преобразовать строку **Hello World** в шестнадцатиричный вид, либо изменить тип данных с **VARCHAR2** на **RAW** — изменить только тип данных, не меняя сами байты данных. Например:

```
scott@DEV816>update demo
  2    set theBlob = utl_raw.cast_to_raw('Hello World')
  3   where id = 1
  4  /

1 row updated.

scott@DEV816> commit;

Commit complete.

scott@DEV816>select id,
  2       dbms_lob.getlength(theBlob) len,
  3        clean(theBlob) piece,
  4       dbms_lob.substr(theBlob,40,1) raw_data
  5    from demo
  6   where id =1;
      ID   LEN PIECE                RAW_DATA

       1    11 Hello World         48656C6C6F20576F726C64
```

Использование **UTL_RAW.CAST_TO_RAW('Hello World')** обычно намного проще преобразования строки **Hello World** в шестнадцатиричное представление - **48656C6C6F20576F726C64.**

Преобразование данных типа LONG/LONG RAW в большой объект

Преобразовать данные типа **LONG** или **LONG RAW** в большой объект очень просто. Стандартная функция **TO_LOB** языка SQL позволяет это сделать. Использование функции **TO_LOB,** однако, весьма ограничено. Ее можно применять исключительно в операторах **INSERT** или **CREATE TABLE AS SELECT** и только в языке SQL (но не в PL/SQL).

В результате первого ограничения **нельзя** выполнять операторы, подобные следующему:

```
alter table t add column clob_column;
update t set clob_column = to_lob(long_column);
alter table t drop column long_column;
```

При попытке выполнения **UPDATE** будет получено сообщение об ошибке:

```
ORA-00932: inconsistent datatypes
```

Для множественного преобразования типа в существующих таблицах со столбцами **LONG/LONG RAW** придется создавать новую таблицу. В большинстве случаев это вполне допустимо, поскольку данные типа **LONG** и **LONG RAW** хранятся в самой строке (inline) вместе с остальными данными таблицы. Если преобразовать их в большие объекты, а затем удалить столбец типа **LONG,** таблица окажется не в лучшем виде: будет много выделенного и неиспользуемого пространства. Такие таблицы лучше пересоздавать.

Второе ограничение означает, что функцию **TO_LOB** нельзя использовать в PL/SQL-блоке. Чтобы использовать **TO_LOB** в PL/SQL, придется прибегнуть к динамическому SQL. Вскоре я это продемонстрирую.

В следующих примерах мы рассмотрим два способа использования функции **TO_LOB.** Один из них — использование функции **TO_LOB** в операторе CREATE **TABLE AS SELECT** или **INSERT INTO.** Другой способ пригодится, когда данные должны остаться в столбце типа **LONG** или **LONG RAW.** Например, старому приложению нужен именно тип **LONG.** Хотелось бы предоставить другим приложениям возможность работать с этим столбцом как с большим объектом, чтобы можно было обрабатывать его значение в PL/SQL по частям с помощью функций пакета **DBMS_LOB,** например **READ** и SUBSTR.

Начнем с создания данных типа **LONG** и **LONG RAW:**

```
ops$tkyte@DEV816> create table long_table
  2  (id        int primary key,
  3   data       long
  4  )
  5  /

Table created.

ops$tkyte@DEV816> create table long_raw_table
  2  (id       int primary key,
  3   data     long raw
  4  )
  5  /

Table created.

ops$tkyte@DEV816> declare
  2      l_tmp    long := 'Hello World';
  3      l_raw   long raw;
  4  begin
  5     while(length(l_tmp) < 32000)
  6      loop
  7         l_tmp := l_tmp || ' Hello World';
  8      end loop;
  9
 10     insert into long_table
 11     (id, data) values
 12      (1, l_tmp);
 13
 14    l_raw := utl_raw.cast_to_raw(l_tmp);
 15
 16     insert into long_raw_table
 17     (id, data) values
```

```
 18      (1, l_raw);
 19
 20      dbms_output.put_line('created long with length = ' ||
 21                           length(l_tmp));
 22  end;
 23  /
created long with length = 32003

PL/SQL procedure successfully completed.
```

Пример множественного однократного преобразования типа

Итак, имеется две таблицы с одной строкой и столбцом типа **LONG** или **LONG RAW**. Преобразование типа данных из **LONG** в **CLOB** легко выполнить с помощью следующего оператора **CREATE TABLE AS SELECT**:

```
ops$tkyte@DEV816> create table clob_table
  2  as
  3  select id, to_lob(data) data
  4    from long_table;

Table created.
```

Кроме того, мы могли создать таблицы ранее и использовать для наполнения ее данными разновидность оператора INSERT INTO:

```
ops$tkyte@DEV816> insert into clob_table
  2  select id, to_lob(data)
  3    from long_table;

1 row created.
```

Следующий пример показывает, что функция TO_LOB не работает в PL/SQL-блоке, как и следовало ожидать:

```
ops$tkyte@DEV816> begin
  2      insert into clob_table
  3      select id, to_lob(data)
  4        from long_table;
  5  end;
  6  /
begin
*
ERROR at line 1:
ORA-06550: line 3, column 16:
PLS-00201: identifier 'TO_LOB' must be declared
ORA-06550: line 2, column 5:
PL/SQL: SQL Statement ignored
```

Это ограничение легко обойти с помощью динамического SQL (придется выполнять оператор **INSERT** динамически, а не статически, как в примере выше). Теперь, разобравшись, как преобразовывать данные типа **LONG** или **LONG RAW** в тип **CLOB** или **BLOB**, рассмотрим производительность такого преобразования. Обычно таблицы со

столбцами типа **LONG** и **LONG RAW** — большого размера. Они большие по определению, поскольку используются для хранения очень больших объектов. Во многих случаях их размер достигает многих гигабайт. Вопрос в том, можно ли выполнить множественное преобразование за допустимое время? Рекомендую использовать следующие возможности:

- невосстановимые действия, такие как непосредственная вставка и опция **NOLOGGING;**
- распараллеливание операторов ЯМД (в частности, параллельные вставки);
- параллельные запросы.

Ниже представлен пример использования этих возможностей. У меня есть большая таблица **IMAGE,** содержащая многие сотни загруженных из Web файлов. Таблица содержит столбцы NAME (название документа), **MIME_TYPE** (например, **application/MS-Word), IMG_SIZE** (размер документа в байтах) и, наконец, сам документ в столбце типа and **LONG RAW.** Преобразуем эту таблицу так, чтобы документ хранился в столбце типа **BLOB.** Можно начать с создания новой таблицы:

```
scott@DEV816>CREATE TABLE "SCOTT"."T"
  2  ("NAME" VARCHAR2(255),
  3   "MIME_TYPE"  VARCHAR2(255),
  4   "IMG_SIZE" NUMBER,
  5    "IMAGE" BLOB)
  6  PCTFREE 0 PCTUSED 40
  7  INITRANS 1
  8  MAXTRANS 255
  9 NOLOGGING
 10  TABLESPACE "USERS"
 11  LOB ("IMAGE") STORE AS
 12   (TABLESPACE "USERS"
 13   DISABLE STORAGE IN ROW CHUNK 32768
 14   PCTVERSION 10
 15   NOCACHE
 16   NOLOGGING
 17 ) ;
Table created.
```

Обратите внимание, что таблица и большой объект создаются с опцией **NOLOGGING** — это важно. Можно не создавать их так сразу, а применить оператор **ALTER.** Теперь, чтобы преобразовать данные из существующей таблицы **IMAGE,** выполним следующее:

```
scott@DEV816>  ALTER SESSION ENABLE PARALLEL DML;
Session altered.
scott@DEV816>  INSERT /*+ APPEND PARALLEL(t,5) */ INTO t
  2   SELECT /*+ PARALLEL(long_raw,5) */
  3               name, mime_type, img_size, to_lob(image)
  4     FROM long_raw;
```

В результате выполняется непосредственная параллельная вставка в объекты типа **BLOB** без журнализации. Для сравнения я выполнил **INSERT INTO** с включенной и

отключенной журнализацией и получил следующие результаты (на подмножестве преобразуемых строк):

```
scott@DEV816> create table t
  2  as
  3  select name, mime_type, img_size, to_lob(image) image
  4  from image where l=0;
Table created.

scott@DEV816> set autotrace on

scott@DEV816> insert into t
  2  select name, mime_type, img_size, to_lob(image) image
  3  from image;
99 rows created.

Execution Plan
----------------------------------------------------------
   0      INSERT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (FULL) OF 'IMAGE'

Statistics
----------------------------------------------------------
       1242  recursive calls
      36057  db block gets
      12843  consistent gets
       7870  physical reads
   34393500  redo size
       1006  bytes sent via SQL*Net to client
        861  bytes received via SQL*Net from client
          4  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
         99  rows processed
```

Обратите внимание, что в результате было сгенерировано 34 Мбайт данных повторного выполнения (если суммировать размеры 99 изображений, получится 32 Мбайт данных). Если таблица T, как было показано ранее, создается с опцией **NOLOGGING** и используется непосредственная вставка, получим:

```
scott@DEV816> INSERT /*+ APPEND */ INTO t
  2  SELECT name, mime_type, img_size, to_lob(image)
  3    FROM image;

99 rows created.

Execution Plan

   0      INSERT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (FULL) OF 'IMAGE'

Statistics

       1242  recursive calls
      36474  db block gets
      13079  consistent gets
       6487  physical reads
```

```
1355104  redo size
   1013  bytes sent via SQL*Net to client
    871  bytes received via SQL*Net from client
      4  SQL*Net roundtrips to/from client
      2  sorts (memory)
      0  sorts (disk)
     99  rows processed
```

Сгенерировано лишь около 1 Мбайт информации в журнал. Это преобразование выполняется существенно быстрее, при этом генерируется намного меньше данных в журналы повторного выполнения. Конечно, как и для всех невосстанавливаемых операций, необходимо обеспечить резервное копирование базы данных как можно раньше, чтобы новые объекты можно было восстановить. Иначе в случае сбоя диска преобразование данных придется выполнять заново.

Представленный выше пример нельзя повторить непосредственно. У меня случайно под рукой оказалась таблица ***IMAGE,*** *содержащая около 200 Мбайт данных. Она использовалась для демонстрации множественных однократных преобразований и влияния опции* ***NOLOGGING*** *на объем генерируемых при этом данных повторного выполнения.*

Оперативное преобразование типа данных

Во многих случаях необходимо читать данные типа **LONG** или **LONG RAW** в различных средах, но оказывается, что это не получается. Например, при использовании языка PL/SQL, если объем данных типа **LONG RAW** превышает 32 Кбайт, их практически невозможно прочитать. В других языках и интерфейсах тоже есть проблемы с данными типа **LONG** и **LONG RAW.** С помощью функции **TO_LOB** и временной таблицы, однако, можно оперативно преобразовать данные типа **LONG** или **LONG RAW** в тип **CLOB** или **BLOB.** Это очень удобно, например, при использовании средств загрузки файлов в OAS4.x или WebDB. Эти средства загружают документы по сети (через Web) в таблицу базы данных, но, к сожалению, загружают они их в столбец типа **LONG RAW.** Это делает практически невозможной работу с документами в PL/SQL. Представленные ниже функции показывают, как обеспечить прозрачный доступ к таким данным через промежуточный **BLOB**-объект.

Начнем с создания временной таблицы для хранения преобразованного объекта типа **CLOB/BLOB** и последовательности, идентифицирующей строку:

```
ops$tkyte@DEV816> create global temporary table lob_temp
  2  (id    int primary key,
  3   c_lob clob,
  4   b_lob blob
  5  )
  6  /

Table created.

ops$tkyte@DEV816> create sequence lob_temp_seq;

Sequence created.
```

Теперь создадим функции **TO_BLOB** и **TO_CLOB.** Эти функции используют для оперативного преобразования данных типа **LONG или LONG RAW** следующий подход.

- Пользователь выбирает идентификатор строки из таблицы со столбцом типа LONG или **LONG RAW,** а не значение столбца **LONG** или **LONG RAW** в этой строке. Функции передается имя столбца типа **LONG,** имя таблицы и идентификатор нужной строки.
- Функция получает последовательный номер, идентифицирующий строку, которая будет создаваться во временной таблице.
- С помощью динамического SQL к указанному столбцу типа **LONG** или **LONG RAW** применяется функция **TO_LOB.** Использование динамического SQL не только делает функцию универсальной (она может работать со столбцом типа **LONG** в любой таблице), но и позволяет непосредственно вызывать функцию **TO_LOB** в языке PLSQL.
- Функция считывает из временной значение созданного объекта типа **BLOB** или **CLOB** таблицы и возвращает вызывающему.

Вот код для функций **TO_BLOB** и **TO_CLOB:**

```
ops$tkyte@DEV816> create or replace
  2  function to_blob(p_cname in varchar2,
  3                   p_tname in varchar2,
  4                   p_rowid in rowid)  return blob
  5  as
  6      l_blob blob;
  7      l_id   int;
  8  begin
  9     select lob_temp_seq.nextval into l_id from dual;
 10
 11     execute immediate
 12        'insert into lob_temp (id,b_lob)
 13        select :id, to_lob(' || p_cname || ')
 14          from ' || p_tname ||
 15        ' where rowid = :rid '
 16      using IN l_id, IN p_rowid;
 17
 18     select b_lob into l_blob from lob_temp where id - l_id ;
 19
 20     return l_blob;
 21  end;
 22  /

Function created.

ops$tkyte@DEV816> create or replace
  2  function to_clob(p_cname in varchar2,
  3          p_tname          in varchar2,
  4                  p_rowid in rowid)  return clob
  5  as
  6      l_clob clob;
  7      l_id   int;
```

```
  8  begin
  9      select lob_temp_seq.nextval into l_id from dual;
 10
 11      execute immediate
 12         'insert into lob_temp (id,c_lob)
 13        select :id, to_lob(' || p_cname || ')
 14          from ' || p_tname ||
 15        ' where rowid = :rid '
 16       using IN l_id, IN p_rowid;
 17
 18      select c_lob into l_clob from lob_temp where id = l_id ;
 19
 20      return l_clob;
 21  end;
 22  /

Function created.
```

Теперь можно продемонстрировать использование этих функций с помощью простого PL/SQL-блока. Данные типа LONG RAW в BLOB будут преобразованы, и выдана длина полученного объекта и небольшая часть его данных:

```
ops$tkyte@DEV816> declare
  2      l_blob    blob;
  3      l_rowid rowid;
  4  begin
  5      select rowid into l_rowid from long_raw_table;
  6      l_blob :=to_blob('data', 'long_raw_table', l_rowid);
  7      dbms_output.put_line(dbms_lob.getlength(l_blob));
  8      dbms_output.put_line(
  9           utl_raw.cast_to_varchar2(
 10               dbms_lob.substr(l_blob,41/1)
 11                                    )
 12                           );
 13  end;
 14  /
32003
Hello World Hello World Hello World Hello

PL/SQL procedure successfully completed.
```

Для тестирования функции **TO_CLOB** применяется практически такой же код, но использовать средства пакета **UTL_RAW** не нужно:

```
ops$tkyte@DEV816> declare
  2      l_clob    clob;
  3      l_rowid rowid;
  4  begin
  5      select rowid into l_rowid from long_table;
  6      l_clob :=to_clob('data', 'long_table', l_rowid);
  7      dbms_output.put_line(dbms_lob.getlength(l_clob));
  8      dbms_output.put_line(dbms_lob.substr(l_clob,41,1));
  9  end;
 10  /
```

```
32003
Hello World Hello World Hello World Hello
PL/SQL procedure successfully completed.
```

Запись значений объекта типа BLOB/CLOB на диск

Этой возможности в пакете **DBMS_LOB** недостает. Пакет предоставляет средства загрузки больших объектов из файлов, но не создания файла, содержащего большой объект. Решение этой проблемы предложено в главах 18 и 19. Там приведен код на языке C и Java для внешней процедуры, записывающей значение столбца типа **BLOB, CLOB** в базе данных или временного большого объекта в файл файловой системы сервера. Обе реализации выполняют одну и ту же функцию, просто использованы разные языки. Применяйте ту из них, которая больше подходит для вашего сервера (например, если на сервере не установлена поддержка языка Java, но есть прекомпилятор Pro*C и компилятор языка C, то внешняя процедура на языке C подойдет больше).

Выдача большого объекта на Web-странице с помощью PL/SQL

Представленный ниже пример предполагает, что в системе установлены и работают следующие компоненты:

- компонент прослушивания (lightweight listener) WebDB;
- сервер приложений OAS 2.x, 3.x или 4.x с PL/SQL-картриджем;
- сервер iAS с модулем **mod_plsql.**

При отсутствии любого из этих компонентов пример выполнить не получится. В нем используется набор инструментальных средств PL/SQL Web Toolkit (речь идет о широко известных функциях **HTP),** а также PL/SQL-картридж или модуль.

Предполагается также, что наборы символов *(кодировки)* на Web-сервере (клиенте сервера базы данных) и в базе данных совпадают. Дело в том, что PL/SQL-картридж или модуль использует для генерации страниц из базы данных тип **VARCHAR2.** Если набор символов у клиента (в данном случае клиентом является Web-сервер) отличается от набора символов в базе данных, будет выполнено преобразование. При этом обычно повреждаются данные типа **BLOB.** Предположим, Web-сервер работает на платформе Windows NT. Обычно для клиента на платформе Windows NT используется набор символов **WE8ISO8859P1** — западноевропейская 8-битовая кодовая страница. А сервер баз данных работает на платформе Solaris. Стандартной и наиболее типичной кодовой страницей на этой платформе является 7-битовая **US7ASCII.** При попытке передачи значения **BLOB** через интерфейс **VARCHAR2** в случае использования такой пары кодовых страниц окажется, что старший бит данных из базы сброшен. Данные изменятся. Только если кодировки на клиенте (Web-сервере) и сервере базы данных совпадают, данные передаются без искажений.

Итак, предполагая, что все предварительные условия выполнены, можно рассмотреть использование средств PL/SQL Web Toolkit для выдачи значения **BLOB** на Web-странице. Продолжим один из предыдущих примеров преобразования, в котором была создана таблица **DEMO**. Загрузим в нее еще один файл:

```
ops$tkyte@DEV816> exec load_a_file('MY_FILES', 'demo.gif');

PL/SQL procedure successfully completed.
```

Это будет GIF-файл. Теперь необходим пакет, который сможет выбрать это изображение в формате GIF и выдать его на Web-странице. Он может иметь следующий вид:

```
ops$tkyte@DEV816> create or replace package image_get
  2  as
  3       — Можно задать соответствующее имя процедуры
  4       — для каждого типа отображаемых документов,
  5       — например:
  6       — procedure pdf
  7       — procedure doc
  8       — procedure txt
  9      — и т.д. Некоторые браузеры (MS IE, например) при обработке
 10       — документов используют расширения имен файлов,
 11       — а не mime-типы
 12      procedure gif(p_id in demo.id%type);
 13  end;
 14  /

Package created.

ops$tkyte@DEV816> create or replace package body image_get
  2  as
  3
  4  procedure gif(p_id in demo.id%type)
  5  is
  6       l_lob   blob;
  7       l_amt   number default 32000;
  8       l_off   number default 1;
  9       l_raw   raw(32000);
 10  begin
 11
 12       — Получить LOB-локатор для
 13       — нашего документа.
 14       select theBlob into l_lob
 15         from demo
 16        where id = p_id;
 17
 18       — Выдать mime-заголовок для
 19       — документа этого типа.
 20      owa_util.mime_header('image/gif');
 21
 22      begin
 23          loop
 24             dbms_lob.read(l_lob, l_amt, l_off, l_raw);
 25
```

```
26            — Важно использовать вызов htp.PRN, чтобы избежать
27             — добавления в документ ненужных символов
28              — перевода строки.
29           htp.prn(utl_raw.cast_to_varchar2(l_raw));
30             l_off  := l_off+l_amt;
31             l_amt  := 32000;
32         end loop;
33     exception
34        when no_data_found then
35             NULL;
36     end;
37 end;
38
39 end;
40 /
Package body created.
```

При наличии DAD (Database Access Descriptor — дескриптор доступа к базе данных, который обычно создается при настройке PL/SQL-картриджа или модуля) с именем mydata можно использовать адрес URL

```
http://myhost:myport/pls/mydata/image_get.gif?p_id=3
```

для получения изображения. Аргумент **P_ID=3** передается процедуре **image_get.gif,** требующий от нее выдать локатор большого объекта, который хранится в строке со значением **id=3.** Это изображение можно включить в страницу с помощью тэга **IMG:**

```
<html>
<head><title>ЭТО моя страница</title></head>
<body>
Это мой GIF-файл
<img src=http://myhost:myport/pls/mydata/image_get.gif?p_id=3>
</body>
</htrol>
```

Резюме

Большие объекты предлагают намного больше возможностей, чем устаревший тип данных **LONG.** В этом разделе я ответил на некоторые часто задаваемые вопросы, касающиеся работы с большими объектами. Мы рассмотрели, как загружать большие объекты в базу данных. Мы разобрались, как преобразовать данные типа **BLOB** в **CLOB,** и наоборот. Мы выяснили, как эффективно преобразовать все существующие унаследованные данные типа **LONG** и **LONG RAW** в типы **CLOB** и **BLOB** с помощью невосстановимых и распараллеливаемых действий. Наконец, мы обсудили использование средств PL/SQL Web Toolkit для получения данных типа **CLOB** или **BLOB** и отображения их на Web-странице.

Пакет DBMS_LOCK

Пакет **DBMS_LOCK** дает программисту доступ к механизму блокирования, используемому сервером Oracle. Он позволяет создавать собственные именованные блокировки. Эти блокировки можно контролировать точно так же, как и любые другие блокировки Oracle. Они будут отображаться в представлении динамической производительности **V$LOCK** как блокировки типа **UL** (user lock — пользовательская блокировка). Кроме того, они будут отображаться любыми стандартными средствами, такими как Oracle Enterprise Manager и сценарий **UTLOCKT.SQL** (который находится в каталоге **[ORACLE_HOME]/rdbms/admin).** Помимо обеспечения доступа к механизму блокирования пакет **DBMS_LOCK** (благодаря наличию функции **SLEEP)** позволяет приостановить работу PL/SQL-программы на указанное количество секунд.

Пакет **DBMS_LOCK** имеет много применений, например.

- Предположим, имеется подпрограмма, использующая средства пакета **UTL_FILE** для записи сообщений проверки в файл операционной системы. Записывать сообщения в этот файл процессы должны поочередно. В некоторых операционных системах, например в ОС Solaris, записывать данные в файл могут одновременно много пользователей (ОС этого не предотвращает). В результате сообщения с данными проверки перемешиваются, и читать их сложно или невозможно. Пакет **DBMS_LOCK** можно использовать для обеспечения очередности доступа к этому файлу.
- Можно предотвратить одновременное выполнение взаимоисключающих действий. Предположим, имеется программа подготовки данных, которая работает только при условии, что данные не используются другими сеансами. Сеансы не должны

обращаться к данным в процессе их подготовки. Сеанс подготовки должен устанавливать именованную блокировку в режиме X (как исключительную). Другие сеансы должны пытаться установить эту же именованную блокировку в режиме S (как разделяемую). Запрос блокировки X будет ожидать, если имеются блокировки S, а запрос блокировки S будет ожидать, если удерживается блокировка X. В результате сеанс подготовки данных будет в состоянии ожидания, пока работают "обычные" сеансы, но если сеанс подготовки уже начался, все остальные сеансы будут заблокированы до его завершения.

У этого пакета есть два основных варианта использования. Оба они подходят, если все сеансы согласованно используют блокировки (ничто не мешает сеансу использовать средства пакета UTL_FILE для открытия и записи в файл проверки без каких-либо попыток установить соответствующую блокировку). В качестве примера попытаемся решить проблему взаимоисключающего доступа, что пригодится во многих приложениях. Проблема возникает при попытке двух сеансов вставить данные в одну и ту же таблицу, для которой задано требование первичного ключа или уникальности. Если оба сеанса попытаются использовать одни и те же значения в столбцах, связанных этим требованием, второй (третий и т.д.) сеанс будет заблокирован, пока не зафиксируется или отменится транзакция первого сеанса. Когда первый сеанс зафиксирует транзакцию, в заблокированных сеансах будет получено сообщение об ошибке. Только если в первом сеансе будет выполнен откат, один из последующих сеансов сможет успешно выполнить вставку. Суть проблемы в том, что пользователи после вынужденного ожидания узнают, что выполнить необходимое Действие невозможно.

Этой проблемы можно избежать при использовании оператора **UPDATE**, поскольку можно заранее заблокировать строку, которую предполагается менять, так, чтобы работа других сеансов не блокировалась. Другими словами, вместо выполнения:

```
update emp set ename = 'King' where empno = 1234;
```

можно написать:

```
select ename from emp where empno = 1234 FOR UPDATE NOWAIT;
update emp set ename = 'King' where empno = 1234;
```

За счет использования конструкции **FOR UPDATE NOWAIT** в операторе **SELECT** можно заблокировать строку для использования сеансом (так что выполнение **UPDATE** не будет заблокировано) или будет получено сообщение об ошибке **ORA-54 'Resource Busy'.** Если при выполнении оператора **SELECT** сообщений об ошибках не получено, строка уже заблокирована.

Однако при выполнении операторов **INSERT** этот метод неприменим. Нет строки, которую можно было бы выбрать с помощью **SELECT** и заблокировать, а потому нет и способа предотвратить вставку строки с таким же значением в других сеансах, что приведет к блокированию и потенциально бесконечному ожиданию в текущем сеансе. Вот тут и поможет пакет **DBMS_LOCK.** Чтобы продемонстрировать, как, я создам таблицу с первичным ключом, предотвращающим одновременную вставку одних и тех же значений двумя (или более) сеансами. Для этой таблицы я задам триггер. Триггер будет использовать функцию **DBMS_UTILITY.GET_HASH_VALUE** (подробнее о ней см. в разделе, посвященном пакету **DBMS UTILITY,** далее в этом приложении) для получе-

ния по первичному ключу числового хеш-значения в диапазоне от 0 до 1073741823 (диапазон значений идентификаторов блокировок, допускаемых сервером Oracle). В этом примере я задал размер хеш-таблицы равным 1024, т.е. по первичным ключам будет получено одно из 1024 значений идентификаторов блокировок. Затем я использую вызов **DBMS_LOCK.REQUEST** для выделения исключительной блокировки с этим идентификатором. В каждый момент времени это сможет сделать только один сеанс, поэтому, если другой сеанс попытается вставить запись в таблицу с таким же первичным ключом, его запрос на блокировку завершится неудачно (и будет получено сообщение об ошибке **RESOURCE BUSY**):

```
tkyte@TKYTE816> create table demo (x int primary key);

Table created.

tkyte@TKYTE816> create or replace trigger demo_bifer
  2  before insert on demo
  3  for each row
  4  declare
  5      l_lock_id   number;
  6      resource_busy   exception;
  7     pragmaexception_init(resource_busy, -54);
  8  begin
  9      l_lock_id :=
 10         dbms_utility.get_hash_value(to_char(:new.x), 0, 1024);
 11
 12      if (dbms_lock.request
 13               (id                => l_lock_id,
 14                lockmode          => dbms_lock.x_mode,
 15                timeout           => 0,
 16          release_on_commit       => TRUE) = 1)
 17      then
 18          raise resource_busy;
 19      end if;
 20  end;
 21  /

Trigger created.
```

Если в двух отдельных сеансах теперь выполнить:

```
tkyte@TKYTE816> insert into demo values (1);

1 row created.
```

то в первом сеансе оператор выполнится, но во втором будет выдано:

```
tkyte@TKYTE816> insert into demo values (1);
insert into demo values (1)
            *
ERROR at line 1:
ORA-00054: resource busy and acquire with NOWAIT specified
ORA-06512: at "TKYTE.DEMO_BIFER", line 15
ORA-04088: error during execution of trigger 'TKYTE.DEMO_BIFER'
```

если в первом сеансе транзакция будет зафиксирована, то будет выдано сообщение о нарушении требования уникальности.

Идея здесь в том, чтобы в триггере брать **первичный ключ** таблицы и помещать его значение в строку символов. После этого можно использовать функцию **DBMS_UTILITY.GET_HASH_VALUE** для получения "почти уникального" хеш-значения для строки. Если использовать хеш-таблицу размером не более 1073741823 значений, можно будет заблокировать это значение в исключительном режиме с помощью пакета **DBMS_LOCK.** Можно также использовать подпрограмму ALLOCATE_UNIQUE пакета **DBMS_LOCK,** но на это потребуются дополнительные ресурсы. Подпрограмма **ALLOCATE_UNIQUE** создает уникальный идентификатор блокировки в диапазоне от 1073741824 до 1999999999. Для этого она использует другую таблицу в базе данных и рекурсивную (автономную) транзакцию. Благодаря хешированию используется меньше ресурсов и, кроме того, можно избежать вызова рекурсивных SQL-операторов.

После хеширования мы берем полученное значение и с помощью пакета **DBMS_LOCK** запрашиваем блокировку с соответствующим идентификатором в исключительном режиме с нулевым временем ожидания (если значение кем-то уже заблокировано, происходит немедленный возврат). Если получить блокировку за это время не удалось, возбуждается исключительная ситуация **ORA-54 RESOURCE BUSY**. В противном случае можно выполнить оператор INSERT, и он не будет заблокирован.

Конечно, если в качестве первичного ключа таблицы используется целое число и есть уверенность, что значения ключа не превысят 1 миллиарда, можно его не хешировать, а использовать непосредственно в качестве идентификатора блокировки.

Нужно "поиграть" с размером хеш-таблицы (в моем примере — 1024), чтобы избежать сообщений **RESOURCE BUSY**, связанных с получением одного и того же хеш-значения по разным строкам. Размер хеш-таблицы зависит от приложения (точнее, от используемых данных); на него также влияет количество одновременно выполняемых вставок. Кроме того, владельцу триггера понадобится непосредственно (не через роль) предоставленная привилегия **EXECUTE** на пакет **DBMS_LOCK**. Наконец, при вставке большого количества строк таким способом, без фиксации может не хватить ресурсов **ENQUEUE_RESOURCES**. В случае возникновения такой проблемы (при этом генерируется соответствующее сообщение) необходимо увеличить значение параметра инициализации **ENQUEUE_RESOURCES**. Можно также добавить в триггер флаг, позволяющий включать и отключать эту проверку. Например, если бы я планировал вставлять сотни/тысячи записей, то не хотел бы выполнять подобную проверку при каждой вставке.

Пользовательские блокировки, а также количество первичных ключей с соответствующим хеш-значением, можно получить из представления **V$LOCK.** Например, если такой триггер был установлен для рассмотренной в предыдущих примерах таблицы **DEMO,** мы получим:

```
tkyte@TKYTE816> insert into demo values (1);

1 row created.

tkyte@TKYTE816> select sid, type, id1
  2    from v$lock
  3   where sid = (select sid from v$mystat where rownum = 1)
  4  /

       SID TY      ID1

         8 TX    589913
```

```
         8 TM      30536
         6 UL        827

tkyte@TKYTE816> begin
  2         dbms_output.put_line
  3            (dbms_utility.get_hash_value(to_char(1),  0,  1024));
  4  end;
  5  /
827

PL/SQL procedure successfully completed.
```

Обратите внимание на пользовательскую блокировку UL со значением ID1 827. Оказывается, что 827 — хеш-значение для результата функции **TO_CHAR**(1), примененной к первичному ключу.

Чтобы завершить этот пример, нужно разобраться, что произойдет, если приложение допускает изменение первичного ключа. В идеале первичный ключ лучше не изменять, но некоторые приложения это делают. Надо учитывать последствия того, если один сеанс изменит значение первичного ключа:

```
tkyte@TKYTE816> update demo set x = 2 where x = 1;
1 row updated.
```

а другой сеанс попытается вставить строку с измененным значением первичного ключа:

```
tkyte@TKYTE816> INSERT INTO DEMO VALUES (2);
```

Второй сеанс опять окажется заблокированным. Проблема в том, что не каждый процесс, который может изменить первичный ключ, учитывает измененную схему блокирования. Для решения этой проблемы, связанной с изменением первичного ключа, необходимо изменить событие, вызывающее срабатывание триггера:

```
before insert OR UPDATE OF X on demo
```

Если созданный триггер срабатывает до вставки данных в столбец X или каких-либо изменений его значения, будет происходить именно то, что требуется (и изменение тоже станет неблокирующим).

Резюме

Пакет **DBMS_LOCK** открывает приложениям доступ к внутреннему механизму блокирования сервера Oracle. Как было продемонстрировано, эту возможность можно использоватьдля реализации специфического метода блокирования, расширяющего стандартные возможности. Мы рассмотрели способы использования этого механизма для обеспечения очередности доступа к общему ресурсу (например, к файлу ОС) и для координации конфликтующих процессов. Мы углубленно изучили использование средств пакета **DBMS_LOCK** для предотвращения блокирующих вставок. Этот пример показал особенности использования пакета **DBMS_LOCK**, а также отображение информации о соответствующих блокировках в представлении **V$LOCK**. В завершение я обратил ваше внимание на важность обеспечения координации действий сеансов, связанных со специфическим методом блокирования, описав, как изменение значения первичного ключа может нарушить работу созданного неблокирующего алгоритма вставок.

Пакет DBMS_LOGMNR

Пакеты **LogMiner**, **DBMS_LOGMNR** и **DBMS_LOGMNR_D,** позволяют анализировать файлы журнала повторного выполнения сервера Oracle. Этот анализ может потребоваться в таких случаях:

- Необходимо определить, когда и кем таблица была удалена.
- Необходимо проверить, какие действия выполнялись с таблицей или набором таблиц, чтобы разобраться, кто и что изменял. Такую проверку можно выполнить "постфактум". Обычно достаточно выполнить команду **AUDIT** (но ее надо выполнять заранее), и вы узнаете, что кто-то изменил таблицу, а вот что именно было изменено узнать таким образом невозможно. Пакеты LogMiner позволяют определить постфактум лицо, внесшее изменения, и какие именно данные были изменены.
- Необходимо "отменить" транзакцию. Для этого надо узнать, что было сделано в этой транзакции, и создать PL/SQL-код для отмены этих действий.
- Необходимо получить эмпирические значения количества строк, изменяемых типичной транзакцией.
- Необходимо выполнить ретроспективный анализ использования базы данных за определенный период времени.
- Необходимо определить, почему сервер вдруг стал генерировать в журнал по 10 Мбайт данных в минуту. Можно ли найти очевидные причины при беглом просмотре журналов?

- Необходимо разобраться, что в действительности происходит "за кадром". Содержимое журналов повторного выполнения показывает, что именно происходит при выполнении вставки в таблицу с триггером, изменяющим другую таблицу. Все результаты транзакции записываются в журнал. Пакеты LogMiner — прекрасное средство изучения этих результатов.

Пакеты LogMiner предоставляют средства для решения всех этих и многих других задач. В этом разделе я представлю краткий обзор использования пакетов LogMiner, а затем опишу ряд проблем при его использовании, о которых не сказано в руководстве *Supplied PL/SQL Packages Reference,* поставляемом корпорацией Oracle. Как и в случае прочих пакетов, рекомендуется прочитать разделы руководства *Supplied PL/SQL Packages Reference,* посвященные пакетам **DBMS_LOGMNR** и **DBMS_LOGMNR_D,** чтобы получить общее представление о функциях и процедурах, которые они содержат, и о принципах использования этих пакетов. Далее в разделе "Опции и использование" представлен обзор соответствующих процедур и их входных данных.

Пакеты LogMiner лучше всего работают с архивными файлами журнала повторного выполнения, хотя их можно использовать и для анализа неактивных оперативных файлов журнала повторного выполнения. Попытка анализа активного оперативного файла журнала повторного выполнения может привести к выдаче сообщения об ошибке или дать некорректные результаты, поскольку файл журнала повторного выполнения содержит данные старых и новых транзакций. Интересно, что с помощью LogMiner можно анализировать файл журнала, первоначально созданный в другой базе данных. Даже версии серверов при этом могут не совпадать (архивные файлы версии 8.0 можно анализировать на сервере версии 8.1). Можно перенести архивный файл журнала повторного выполнения в другую систему и анализировать его там. Это весьма удобно в случае проверки или ретроспективного анализа тенденций использования, не влияющей на работу системы. Для этого, однако, надо использовать сервер на той же аппаратной платформе (т.е. обеспечить тот же порядок байтов, размер слова и т.п.). Желательно обеспечить такой же размер блоков базы данных, как в исходной (размер блока в базе данных, где выполняется анализ, **не должен быть меньше,** чем в базе данных, где журнал повторного выполнения сгенерирован), и совпадение кодовых страниц.

Процесс использования пакетов LogMiner состоит из двух этапов. На первом — создается словарь данных для работы пакетов LogMiner. Именно это и позволяет анализировать файл журнала повторного выполнения не в той базе данных, где он был сгенерирован (пакеты LogMiner не используют существующий словарь данных). Используется словарь данных, экспортированный во внешний файл с помощью пакета **DBMS_LOGMNR_D.** Пакеты LogMiner можно использовать и без этого словаря данных, но разобраться в полученных результатах при этом практически невозможно. Формат представления этих результатов мы рассмотрим несколько позже.

На втором этапе импортируются файлы журнала повторного выполнения, и запускается LogMiner. После запуска основного пакета LogMiner можно просматривать содержимое файлов журнала повторного выполнения с помощью SQL-операторов. С пакетами LogMiner связано четыре представления V$. Основное представление — **V$LOGMNR_CONTENTS.** Именно оно будет использоваться для анализа содержимого загруженных файлов журнала повторного выполнения. Более детально это представ-

ление мы рассмотрим в примере, а в конце раздела представлена таблица с описанием его столбцов. Остальные три представления описаны ниже.

- **V$LOGMNR_DICTIONARY.** Это представление содержит информацию о загруженном файле словаря. Этот словарь был создан на первом этапе. Чтобы разобраться в содержимом файла журнала повторного выполнения, необходим файл словаря, задающий имя объекта с данным идентификатором, имена и типы данных столбцов таблиц и т.п. Это представление содержит не больше одной строки, описывающей текущий загруженный словарь.
- **V$LOGMNR_LOGS.** Это представление содержит информацию о файлах журнала повторного выполнения, которые пользователь загрузил в систему с помощью LogMiner. Содержимое файлов журнала повторного выполнения можно найти в представлении **V$LOGMNR_CONTENTS.** Там же вы найдете характеристики самих файлов: имя файла журнала повторного выполнения, имя базы данных, в которой он сгенерирован, номера системных изменений (**SCNs** — system change numbers), содержащихся в нем, и т.д. В представлении содержится по одной строке для каждого анализируемого файла.
- **V$LOGMNR_PARAMETERS.** Это представление содержит параметры, переданные LogMiner при запуске. После вызова подпрограммы запуска LogMiner в нем будет одна строка.

Важно отметить, что, поскольку пакеты LogMiner выделяют память в области PGA, средства LogMiner нельзя использовать в среде MTS. Дело в том, что в среде MTS каждое обращение к базе данных будет обрабатываться другим разделяемым сервером (процессом или потоком). Данные, загруженные первым процессом (первым разделяемым сервером) не доступны для второго процесса (второго разделяемого сервера). Для работы пакетов LogMiner необходимо подключиться к выделенному серверу. Кроме того, результат доступен в одном сеансе и только в процессе его работы. Если необходим дальнейший анализ, надо либо загрузить информацию повторно, либо сохранить ее в постоянной таблице, например, с помощью оператора **CREATE TABLE AS SELECT.** При анализе больших объемов данных размещение их в обычной таблице с помощью операторов **CREATE TABLE AS SELECT** или **INSERT INTO** имеет еще больше смысла. В дальнейшем эту информацию можно проиндексировать, тогда как представление **V$LOGMNR_CONTENTS** всегда просматривается полностью, что требует очень много ресурсов.

Обзор

После обзора средств LogMiner мы рассмотрим назначение входных параметров стандартных пакетов LogMiner. Затем я расскажу вам, как с помощью LogMiner определить, когда в базе данных произошло определенное действие. Далее будет рассмотрено влияние пакетов LogMiner на использование памяти сеансами, а также кэширование пакетами файлов журнала повторного выполнения. Наконец, я опишу ограничения, связанные с использованием пакетов LogMiner, не упоминающиеся в документации.

Этап 1: создание словаря данных

Чтобы средства LogMiner могли сопоставить внутренним идентификаторам объектов и столбцов соответствующие имена, необходим словарь данных. Имеющийся в базе данных словарь при этом не используется. Словарь данных должен загружаться из внешнего файла. Это необходимо для того, чтобы журналы повторного выполнения можно было анализировать в другой базе данных. Кроме того, текущий словарь данных в базе может поддерживать уже не все объекты, находившиеся в базе данных в момент генерации файла журнала повторного выполнения, вот почему словарь данных необходимо импортировать.

Чтобы понять назначение файла словаря данных, давайте рассмотрим результата работы LogMiner, когда словарь данных не загружен. Для этого загрузим архивный файл журнала повторного выполнения и запустим LogMiner. Затем выполним запрос к представлению **V$LOGMNR_CONTENTS**, чтобы определить его содержимое:

```
tkyte@TKYTE816> begin
  2    sys.dbms_logmnr.add_logfile
  3   ('C:\oracle\oradata\tkyte816\archive\TKYTE816T001S01263.ARC',
  4        sys.dbms_logmnr.NEW);
  5  end;
  6  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> begin
  2    sys.dbms_logmnr.start_logmnr;
  3  end;
  4  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816>column sql_redo format a30
tkyte@TKYTE816>column sql_undo format a30
tkyte@TKYTE816>select scn, sql_redo, sql_undo from v$logmnr_contents
  2  /

       SCN SQL_REDO                       SQL_UNDO

6.4430E+12
6.4430E+12 set transaction read write;
6.4430E+12 update UNKNOWN.Objn:30551 set  update UNKNOWN.Objn:30551 set
           Col[2] = HEXTORAW('787878') wh Col[2] = HEXTORAW('534d495448'
           ere ROWID = 'AAAHdXAAGAAAAJKAA ) where ROWID = 'AAAHdXAAGAAAA
           A';                            JKAAA';

6.4430E+12
6.4430E+12 commit;
```

```
tkyte@TKYTE816> select utl_raw.cast_to_varchar2(hextoraw('787878')) from dual;

UTL_RAW.CAST_TO_VaRCHAR2(HEXTORAW('787878'))

XXX
```

```
tkyte@TKYTE816> select utl_raw.cast_to_varchar2(hextoraw('534d495448'))
                                                              from dual;

UTL_RAW.CAST_TO_VARCHAR2(HEXTORAW('534D495448•))

SMITH
```

Читать этот результат практически невозможно. Мы знаем, что изменен столбец 2 объекта с идентификатором 30551. Более того, можно получить значение HEXTORAW('787878') в виде строки символов. Можно обратиться к словарю данных и определить, какой объект имеет идентификатор 30551:

```
tkyte@TKYTE816>select object_name
  2  from all_objects
  3  where data_object_id = 30551;

OBJECT_NAME

EMP
```

но сделать это можно только в той же базе данных, в которой был сгенерирован журнал повторного выполнения, и только в том случае, если этот объект еще существует. Далее можно выполнить команду **DESCRIBE EMP** и определить, что столбец 2 — это **ENAME**. Поэтому в столбце **SQL_REDO** пакет LogMiner поместил значение **UPDATE EMP SET ENAME = 'XXX' WHERE ROWID** = ... К счастью, подобные запутанные преобразования не придется выполнять при каждом анализе журнала. Мы убедимся, что, создав и загрузив словарь, можно получить намного более понятные результаты. Следующий пример показывает, каких результатов можно ожидать, создав файл словаря для пакетов LogMiner, а затем загрузив его.

Начнем с создания файла словаря. Создать его весьма просто. Для этого должны быть выполнены следующие требования.

- Конфигурация параметров инициализации позволяет создавать с помощью пакета **UTL_FILE** файлы хотя бы в одном каталоге. Подробнее соответствующая настройка описана в разделе, посвященном пакету **UTL_FILE**. Пакет **DBMS_LOGMNR_D**, с помощью которого создается файл словаря данных, для выполнения ввода-вывода использует средства пакета **UTL_FILE**.
- Схема, в которой будет вызываться пакет **DBMS_LOGMNR_D**, имеет привилегию **EXECUTE ON SYS.DBMS_LOGMNR_D**, или ей предоставлена роль с привилегий выполнения этого пакета. По умолчанию роль **EXECUTE_CATALOG_ROLE** имеет привилегию для выполнения этого пакета.

После настройки пакета **UTL_FILE** и получения привилегии **EXECUTE ON DBMS_LOGMNR_D** создание файла словаря представляет собой тривиальную задачу. Надо вызвать всего одну подпрограмму пакета **DBMS_LOGMNR_D** — процедуру **BUILD**. Достаточно выполнить примерно следующее:

```
tkyte@TKYTE816>set serveroutput on

tkyte@TKYTE816> begin
  2    sys.dbms_logmnr_d.build('miner_dictionary.dat',
  3                            'c:\temp');
```

```
  4  end;
  5  /
LogMnr Dictionary Procedure started
LogMnr Dictionary File Opened
TABLE: OBJ$ recorded in LogMnr Dictionary File
TABLE: TAB$ recorded in LogMnr Dictionary File
TABLE: COL$ recorded in LogMnr Dictionary File
TABLE: SEG$ recorded in LogMnr Dictionary File
TABLE: UNDO$ recorded in LogMnr Dictionary File
TABLE: UGROUP$ recorded in LogMnr Dictionary File
TABLE: TS$ recorded in LogMnr Dictionary File
TABLE: CLU$ recorded in LogMnr Dictionary File
TABLE: IND$ recorded in LogMnr Dictionary File
TABLE: ICOL$ recorded in LogMnr Dictionary File
TABLE: LOB$ recorded in LogMnr Dictionary File
TABLE: USER$ recorded in LogMnr Dictionary File
TABLE: FILE$ recorded in LogMnr Dictionary File
TABLE: PARTOBJ$ recorded in LogMnr Dictionary File
TABLE: PARTCOL$ recorded in LogMnr Dictionary File
TABLE: TABPART$ recorded in LogMnr Dictionary File
TABLE: INDPART$ recorded in LogMnr Dictionary File
TABLE: SUBPARTCOL$ recorded in LogMnr Dictionary File
TABLE: TABSUBPART$ recorded in LogMnr Dictionary File
TABLE: INDSUBPART$ recorded in LogMnr Dictionary File
TABLE: TABCOMPART$ recorded in LogMnr Dictionary File
TABLE: INDCOMPART$ recorded in LogMnr Dictionary File
Procedure executed successfully - LogMnr Dictionary Created

PL/SQL procedure successfully completed.
```

Перед вызовом процедуры **BUILD** пакета **DBMS_LOGMNR_D** рекомендуется выполнять команду SET **SERVEROUTPUT** ON — это обеспечит выдачу информационных сообщений пакета **DBMS_LOGMNR_D.** Они помогут выяснить причины ошибки при выполнении **DBMS_LOGMNR_D.BUILD.** Выполненная выше команда создала файл **C:\TEMP\MINER_DICTIONARY.DAT.** Это обычный текстовый файл, который можно просматривать в текстовом редакторе. Файл содержит SQL-подобные операторы, которые анализируются и выполняются процедурой запуска основного пакета LogMiner. Теперь, при наличии файла словаря, можно посмотреть, какая информация содержится в представлении **V$LOGMNR_CONTENTS:**

```
tkyte@TKYTE816> begin
  2      sys.dbms_logmnr.add_logfile
  3       ('C:\oracle\oradata\tkyte816\archive\TKYTE816T001S01263.ARC',
  4        sys.dbms_logmnr.NEW);
  5  end;
  6  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> begin
  2      sys.dbms_logmnr.start_logmnr
  3    (dictFileName=>    'c:\temp\miner_dictionary.dat');
  4  end;
```

```
  5  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816>column sql_redo format a30
tkyte@TKYTE816>column sql_undo format a30
tkyte@TKYTE816>select scn, sql_redo, sql_undo from v$logmnr_contents
  2  /

       SCN SQL_REDO                       SQL_UNDO

6.4430E+12
6.4430E+12 set transaction read write;
6.4430E+12 update TKYTE.EMP set ENAME = ' update TKYTE.EMP set ENAME = '
           xxx' where ROWID = 'AAAHdXAAGA SMITH' where ROWID = 'AAAHdXAA
           AAAJKAAA';                     GAAAAJKAAA';

6.4430E+12
6.4430E+12 commit;
```

Теперь все гораздо понятнее: можно прочитать SQL-операторы, сгенерированные пакетом LogMiner, и повторяющие (или отменяющие) изучаемую транзакцию. Теперь можно переходить ко второму этапу — использованию средств LogMiner.

Этап 2: использование средств LogMiner

Используем только что сгенерированный файл словаря для анализа содержимого архивных файлов журнала повторного выполнения. Перед загрузкой журнала повторного выполнения сгенерируем такой файл, все транзакции в котором будут известны. Это упростит интерпретацию результатов в первый раз. Мы сможем сопоставлять содержимое представления **V$LOGMNR_CONTENTS** с только что выполненными транзакциями. Для этого важно организовать тестовую базу данных, с которой будет работать только один пользователь. Это позволит искусственно ограничить содержимое журнала повторного выполнения. Для этой базы данных также понадобится привилегия **ALTER SYSTEM,** чтобы можно было принудительно вызвать архивирование файла журнала. Наконец, все гораздо проще, если база данных работает в режиме автоматического архивирования журналов. При этом очень легко найти соответствующий файл журнала повторного выполнения (это будет только что заархивированный файл — ниже я покажу, как его найти). При использовании базы данных в режиме **NOARCHWELOGMODE,** вам придется найти активный журнал и определить, какой файл журнала был активным непосредственно перед ним. Итак, чтобы сгенерировать транзакцию-образец, выполним:

```
tkyte@TKYTE816>alter system archive log current;

System altered.

tkyte@TKYTE816>update emp set ename = lower(ename);

14 rows updated.

tkyte@TKYTE816>update dept set dname = lower(dname);

4 rows updated.
```

```
tkyte@TKYTE816> commit;

Commit complete.

tkyte@TKYTE816> alter system archive log current;

System altered.

tkyte@TKYTE816> column name format a80
tkyte@TKYTE816> select name
  2    from v$archived_log
  3   where completion_time = (select max(completion_time)
  4                              from v$archived_log)
  5  /

NAME

C:\ORACLE\ORADATA\TKYTE816\ARCHrVE\TKYTE816T001S01267.ARC
```

Теперь, с учетом того, что я — единственный пользователь, изменяющий базу данных, в сгенерированном архивном журнале окажутся только два выполненных изменения. Последний запрос к представлению **V$ARCHIVED_LOG** возвращает имя архивного файла журнала повторного выполнения, который необходимо анализировать. Его можно загрузить в LogMiner и анализировать с помощью приведенных ниже PL/SQL-блоков. Они добавят последний архивный файл журнала повторного выполнения к списку обрабатываемых, а затем запустят LogMiner:

```
tkyte@TKYTE816> declare
  2      l_name v$archived_log.name%type;
  3  begin
  4
  5      select name into l_name
  6        from v$archived_log
  7       where completion_time = (select max(completion_time)
  8                                  from v$archived_log);
  9
 10    sys.dbms_logmnr.add_logfile(l_name,   sys.dbms_logmnr.NEW);
 11  end;
 12  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> begin
  2     sys.dbms_logmnr.start_logmnr
  3     (dictFileName => 'c:\temp\miner_dictionary.dat');
  4  end;
  5  /

PL/SQL procedure successfully completed.
```

Первый вызов процедуры, **DBMS_LOGMNR.ADD_LOGFILE,** загрузил архивный файл журнала повторного выполнения в LogMiner. Я передал имя архивного файла журнала повторного выполнения и опцию **DBMS_LOGMNR.NEW.** Поскольку этот файл добавляется первым, надо указывать опцию **DBMS_LOGMNR.NEW.** Поддерживаются также опции **ADDFILE** для добавления еще одного файла журнала в существующий

список файлов, а также **REMOVEFILE** — для удаления файла из списка рассматриваемых. После загрузки нужных файлов журнала можно вызывать процедуру **DBMS_LOGMNR.START_LOGMNR,** передавая ей имя созданного файла словаря. При вызове **START_LOGMNR** передается минимум информации — только имя файла словаря. Другие опции процедуры **START_LOGMNR** мы рассмотрим в разделе "Опции и использование".

Теперь, после загрузки файла журнала и запуска LogMiner, все готово для просмотра содержимого представления **V$LOGMNR_CONTENTS.** Представление **V3LOGMNR_CONTENTS** содержит большое количество информации; для начала рассмотрим только небольшую ее часть. В частности, выберем значения столбцов SCN, **SQL_REDO** и **SQL_UNDO.** Если вы этого еще не знаете, **SCN** (порядковый *номер системного изменения)* — это простой механизм отсчета времени, который сервер Oracle используетдля упорядочения транзакций и восстановления базы данных после сбоев. Эти номера также используются для обеспечения согласованности по чтению и при обработке контрольных точек в базе данных. **SCN** можно рассматривать как счетчик: при фиксации каждой транзакции значение **SCN** увеличивается на единицу. Вот запрос из предыдущего примера, в котором имена в таблицах **EMP** и **DEPT** переводятся в нижний регистр:

```
tkyte@TKYTE816>column sql_redo format a20 word_wrapped
tkyte@TKYTE816>column sql_undo format a20 word_wrapped

tkyte@TKYTE816>select scn, sql_redo, sql_undo from v$logmnr_contents
  2  /

       SCN SQL_REDO             SQL_UNDO

6.4430E+12 set transaction read
           write;

6.4430E+12 update TKYTE.EMP set update TKYTE.EMP set
           ENAME = 'smith'     ENAME - 'SMITH'
           where ROWID =       where ROWID =
           'AAAHdYAAGAAAAJKAAA' 'AAAHdYAAGAAAAJKAAA'


6.4430E+12
6.4430E+12 update TKYTE.EMP set update TKYTE.EMP set
           ENAME = 'allen'     ENAME = 'ALLEN'
           where ROWID =       where ROWID =
           'AAAHdYAAGAAAAJKAAB' 'AAAHdYAAGAAAAJKAAB'


...(несколько аналогичных строк выброшено)...

6.4430E+12 update TKYTE.DEPT   update TKYTE.DEPT
           set DNAME = 'sales'  set DNAME = 'SALES'
           where ROWID =       where ROWID =
           'AAAHdZAAGAAAAKKAAC' 'AAAHdZAAGAAAAKKAAC'
           ;;

6.4430E+12 update TKYTE.DEPT   update TKYTE.DEPT
           set DNAME =         set DNAME =
```

```
 'operations' where   'OPERATIONS' where
ROWID =               ROWID =
 'AAAHdZAAGAAAAKKAAD' 'AAAHdZAAGAAAAKKAAD'
;;

6.4430E+12 commit;

22 rows selected.
```

Как видите, SQL-операторы сгенерировали в журнале повторного выполнения не две строки, а намного больше. Журнал повторного выполнения содержит измененные биты и байты, а не SQL-операторы. Поэтому многострочный оператор **UPDATE EMP SET ENAME = LOWER(ENAME)** представляется средствами LogMiner в виде набора однострочных изменений. В настоящее время LogMiner не позволяет получить реально выполненные SQL-операторы. Можно создать только делающие то же самое SQL-операторы, но в виде набора отдельных операторов.

В этом примере мы пойдем на шаг дальше. В представлении **V$LOGMNR_CONTENTS** есть столбцы-заместители ("placeholder" columns). Столбцы-заместители позволяют найти изменения, выполненные в пяти (но не более) столбцах таблицы. Столбцы-заместители позволяют узнать имя измененного столбца, а также значение столбца до и после изменения. Поскольку эти столбцы выделены из текста SQL-операторов, очень легко найти транзакцию, изменившую, скажем, значение в столбце ENAME (в столбце-заместителе имени будет значение **ENAME**) с **KING** (в столбце предварительного значения будет **KING**) на **king**. Чтобы продемонстрировать это, выполним еще один небольшой пример с оператором **UPDATE** и создадим *файл сопоставления столбцов* (column mapping file). Файл сопоставления столбцов (будем сокращенно называть его файл colmap) позволяет указать средствам LogMiner, какие столбцы в таблице вас интересуют. Можно сопоставитьдо пяти столбцов в каждой таблице со столбцами-заместителями. Файл colmap имеет следующий формат:

```
colmap = TKYTE DEPT (1, DEPTNO, 2, DNAME, 3, LOC);
colmap = TKYTE EMP (1, EMPNO, 2, ENAME, 3, JOB, 4, MGR, 5, HIREDATE);
```

При просмотре строки в таблице **DEPT** столбцу **DEPT DEPTNO** будет сопоставлен первый столбец-заместитель. А при просмотре строки в таблице EMP этому столбцу-заместителю будет сопоставлен столбец **EMP EMPNO**.

Файл сопоставления столбцов в общем случае состоит из строк следующей структуры (полужирным выделены константы, <sp> представляет обязательный пробел)

```
colmap<sp>-<sp>ВЛАДЕЛЕЦ<sp>ИМЯ_ТАБЛИЦЫ<sp>(1,<sp>ИМЯ_СТОЛБЦА[,<sp>2,<sp>
->ИМЯ СТОЛБЦА] . . . ) ;
```

Регистр символов везде имеет значение: ВЛАДЕЛЬЦА надо задавать в верхнем регистре, имя таблицы — с соблюдением регистра (обычно — в верхнем регистре, если только при создании объекта не использовались идентификаторы в кавычках). Пробелы указывать тоже обязательно. Для того чтобы упростить использование файла сопоставления столбцов я использую следующий сценарий:

```
set linesize 500
set trimspool on
set feedback off
```

```
set heading off
set embedded on
spool logmnr.opt
select
    'colmap= ' || user || ' ' || table_name || ' (' ||
   max(decode(column_id, 1,     column_id  , null)) ||
   max(decode(column_id, 1, ', '||column_name, null)) ||
   max(decode(column_id, 2, ', '||column_id , null)) ||
   max(decode(column_id, 2, ', '||column_name, null)) ||
   max(decode(column_id, 3, ', '||column_id , null)) ||
   max(decode(column_id, 3, ', '||column_name, null)) ||
   max(decode(column_id, 4, ', '||column_id , null)) ||
   max(decode(column_id, 4, ', '||column_name, null)) ||
   max(decode(column_id, 5, ', '||column_id , null)) ||
   max(decode(column_id,5,',','||column_name,null))||' ) ; 'colmap
 from user_tab_columns
group by user, table_name
/
spool off
```

в SQL*Plus для автоматической генерации файла logmnr.opt. Например, если выполнить этот сценарий в схеме, где имеются только таблицы EMP и DEPT, аналогичные тем, что принадлежат пользователю **SCOTT/TIGER**, вы получите:

```
tkyte@TKYTE816> @colmap
colmap = TKYTE DEPT (1, DEPTNO, 2, DNAME, 3, LOC);
colmap = TKYTE EMP (1, EMPNO, 2, ENAME, 3, JOB, 4, MGR, 5, HIREDATE);
```

Я всегда сопоставляю первых пять столбцов таблицы. Если вы хотите использовать другой набор пяти столбцов, отредактируйте полученный при выполнении этого сценария файл **logmnr.opt,** изменив имена столбцов. Например, в таблице **EMP** есть еще три столбца, не представленные в полученном файле colmap — **SAL, COMM** и **DEPTNO.** Если необходимо просматривать изменения столбца **SAL,** а не **JOB,** файл colmap должен выглядеть так:

```
tkyte@TKYTE816> @colmap
colmap = TKYTE DEPT (1, DEPTNO, 2, DNAME, 3, LOC);
colmap = TKYTE EMP (1, EMPNO, 2, ENAME, 3, SAL, 4, MGR, 5, HIREDATE);
```

Помимо использования соответствующего регистра символов и количества пробелов при работе с файлом colmap важно также следующее:

- Файл должен называться **logmnr.opt.** Другое имя использовать нельзя.
- Этот файл должен быть **в том же каталоге,** что и файл словаря.
- Файл **colmap** можно использовать только **вместе с файлом словаря.**

Итак, мы сейчас изменим все столбцы в таблице **DEPT. Я** использую четыре различных оператора **UPDATE,** каждый из которых будет изменять другую строку и набор столбцов. Это позволит увидеть результат сопоставления столбцов-заместителей:

```
tkyte@TKYTE816>alter system archive log current;

tkyte@TKYTE816>update dept set deptno = 11
```

```
  2   where deptno = 40
  3  /

tkyte@TKYTE816> update dept set dname = initcap(dname)
  2   where deptno = 10
  3  /

tkyte@TKYTE816> update dept set loc = initcap(loc)
  2   where deptno = 20
  3  /

tkyte@TKYTE816> update dept set dname = initcap(dname),
  2                     loc = initcap(loc)
  3   where deptno = 30
  4  /

tkyte@TKYTE816> commit;

tkyte@TKYTE816> alter system archive log current;
```

Теперь можно найти изменения в каждом из столбцов, загрузив вновь сгенерированный архивный файл журнала повторного выполнения и запустив LogMiner с опцией **USE_COLMAP**. Обратите внимание, что я сгенерировал файл **logmnr.opt** с помощью представленного ранее сценария и поместил этот файл в тот же каталог, где находится словарь данных:

```
tkyte@TKYTE816> declare
  2      l_name v$archived_log.name%type;
  3  begin
  4
  5      select name into l_name
  6        from v$archived_log
  7       where completion_time = (select max(completion_time)
  8                                  fromv$archived_log);
  9
 10    sys.dbms_logmnr.add_logfile(l_name,   sys.dbms_logmnr.NEW);
 11  end;
 12  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> begin
  2     sys.dbms_logmnr.start_logmnr
  3     (dictFileName => 'c:\temp\miner_dictionary.dat',
  4     options => sys.dbms_logmnr.USE_COLMAP);
  5  end;
  6  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select scn, phl_name, phl_undo, phl_redo,
  2             ph2_name, ph2_undo, ph2_redo,
  3             ph3_name, ph3_undo, ph3_redo
  4    from v$logmnr_contents
  5   where seg_name = 'DEPT'
  6  /
```

```
      SCN PH1_NA PH1 PH1 PH2_N PH2_0NDO  PH2_REDO  PH3 PH3_UNDO PH3_RED0

6.4430E+12 DEPTNO 40  11
6.4430E+12              DNAME accounting Accounting
6.4430E+12                                        LOC DALLAS   Dallas
6.4430E+12              DNAME sales      Sales     LOC CHICAGO  Chicago
```

Итак, этот результат ясно показывает (в первой строке, например), что в столбце **DEPTNO** значение 40 **(PH1)** было изменено на 11. Это понятно, поскольку мы выполняли SET **DEPTNO = 11 WHERE DEPTNO = 40.** Обратите внимание, что остальные столбцы в перовой строке — пустые. Причина в том, что сервер Oracle записывает только измененные байты; для этой строки нет предварительного и окончательного образов столбцов **DNAME** и **LOC.** Вторая строка показывает изменение значения в столбце **DNAME** с **accounting** на **Accounting** и отсутствие изменений в столбцах **DEPTNO** или **LOC,** поскольку они не были затронуты. Последняя строка показывает, что при изменении двух столбцов одним оператором **UPDATE** они будут представлены в столбцах-заместителях.

Как видите, использование столбцов-заместителей может пригодиться при поиске конкретной транзакции в большем объеме данных повторного выполнения. Если известно, что транзакция изменила таблицу X, поменяв в столбце Y значение с a на b, найти ее очень легко.

Опции и использование

Функциональные возможности LogMiner реализуются двумя пакетами - **DBMS_LOGMNR** и **DBMS_LOGMNR_D.** Пакет **DBMS_LOGMNR_D (_D** в названии обозначает "словарь" — "dictionary") содержит всего одну процедуру, **BUILD.** Она применяется для создания словаря данных, используемого пакетом **DBMS_LOGMNR** при загрузке файла журнала повторного выполнения. Он позволяет сопоставить идентификаторам объектов имена таблиц, определить имена и типы данных столбцов по порядковому номеру и т.д. Использовать процедуру **DBMS_LOGMNR_D.BUILD** очень просто. Она имеет два параметра:

- **DICTIONARY_FILENAME.** Имя файла словаря, который необходимо создать. В наших примерах использовался файл **miner_dictionary.dat.**
- **DICTIONARY_LOCATION.** Каталог, в котором этот файл будет создан. Процедура использует для создания файла средства пакета **UTL_FILE,** так что каталог должен быть перечислен среди допустимых каталогов, задаваемых параметром инициализации **utl_file_dir.** Подробнее о конфигурировании пакета **UTL_FILE** см. в соответствующем разделе в конце приложения А.

Вот и все, что нужно знать о процедуре **BUILD.** Оба параметра указывать обязательно. Если при вызове этой процедуры получено сообщение об ошибке, подобное следующему:

```
tkyte@TKYTE816>  exec sys.dbms_logmnr_d.build('x.dat',  'c:\not_valid\');
BEGIN sys.dbms_lognmr_d.build('x.dat', 'c:\not_valid\'); END;
```

```
*
ERROR at line 1:
ORA-01309: specified dictionary file cannot be opened
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at "SYS.DBMS_LOGMNR_D", line 793
ORA-06512: at line 1
```

значит, указанный каталог не указан в параметре инициализации **utl_file_dir.**

Пакет **DBMS_LOGMNR** состоит из трех процедур.

- **ADD_LOGFILE.** Зарегистрировать набор файлов журнала для анализа.
- **START_LOGMNR.** Заполнить данными представление **V$LOGMNR_CONTENTS.**
- **END_LOGMNR.** Освободить все ресурсы, выделенные при работе LogMiner. Эта процедура вызывается для корректного освобождения ресурсов перед завершением сеанса или при окончании работы с пакетами LogMiner.

Процедура **ADD_LOGFILE,** как было сказано ранее, вызывается еще до запуска LogMiner. Она создает список файлов журнала, которые будут обрабатываться при выполнении процедуры **START_LOGMNR** для заполнения представления **VSLOGMNR_CONTENTS.** Процедура **ADD_LOGFILE** принимает следующие параметры:

- **LOGFILENAME.** Полное имя файла архивного журнала повторного выполнения, который необходимо проанализировать.
- **OPTIONS.** Задает, добавлять указанный файл или удалять. В качестве значения задаются следующие константы пакета **DBMS_LOGMNR:**

  DBMS_LOGMNR.NEW. Начать новый список. Если список уже существует, он очищается.

  DBMS_LOGMNR.ADD. Добавить файл в уже существующий или пустой список.

  DBMS_LOGMNR.REMOVEFILE. Удалить файл из списка.

Если необходимо проанализировать последние два архивных файла журнала повторного выполнения, процедура **ADD_LOGFILE** вызывается дважды. Например:

```
tkyte@TKYTE816> declare
  2     l_cnt  number default 0;
  3  begin
  4     for x in (select name
  5                 from v$archived_log
  6                order by completion_time desc)
  7     loop
  8        l_cnt := l_cnt+1;
  9        exit when (l_cnt > 2);
 10
 11        sys.dbms_logmnr.add_logfile(x.name);
 12     end loop;
 13
 14    sys.dbms_logmnr.start_logmnr
 15    (dictFileName => 'c:\temp\miner_dictionary.dat'.
```

```
16        options => sys.dhms_logmnr.USE_COLMAP);
17  end;
18  /

PL/SQL procedure successfully completed.
```

В одном сеансе после запуска LogMiner можно вызывать процедуру **ADD_LOGFILE** для добавления дополнительных файлов журнала, удаления тех из них, которые больше не представляют интереса, или (если указана опция **DBMS_LOGMNR.NEW**) для сброса списка файлов журнала так, чтобы он включал только один указанный новый файл. При вызове **DBMS_LOGMNR.START_LOGMNR** после изменения списка файлов содержимое представления **V$LOGMNR_CONTENTS,** по сути, сбрасывается и создается заново на основе информации в журнальных файлах, входящих в список.

Процедура **DBMS_LOGMNR.START_LOGMNR** принимает много параметров. В рассмотренных ранее примерах мы использовали только два из шести имеющихся. Мы задавали имя файла словаря и опции (чтобы указать, что необходимо использовать файл colmap). В общем случае поддерживаются следующие параметры:

- **STARTSCN и ENDSCN**. Если точно известен диапазон интересующих номеров системных изменений, можно поместить в представление **V$LOGMNR_CONTENTS** только соответствующие строки. Это пригодится после загрузки всего файла журнала и определения максимального и минимального номера системных изменений, которые представляют интерес. Можно перезапустить LogMiner, указав этот диапазон, чтобы уменьшить объем данных в представлении **V$LOGMNR_CONTENTS**. По умолчанию эти параметры имеют значение 0 и не используются.
- **STARTTIME и ENDTIME**. Вместо указания диапазона **SCN** можно задать отрезок времени. Только записи журнала, попадающие в указанный отрезок времени, окажутся в представлении **V$LOGMNR_CONTENTS**. Эти значения игнорируются, если указаны значения **STARTSCN и ENDSCN**. По умолчанию используется отрезок времени с 1 января 1988 года по 1 января 2988 года.
- **DICTFILENAME**. Полное имя файла словаря, созданного процедурой **DBMS_LOGMNR_D.BUILD.**
- **OPTIONS**. В настоящее время поддерживается только одна опция процедуры **DBMS_LOGMNR.START_LOGMNR** - опция **DBMS_LOGMNR.USE_COLMAP**. Она задает поиск файла **logmnr.opt** в том же каталоге, что и файл **DICTFILENAME**. Важно помнить, что файл colmap может иметь только имя **logmnr.opt** и должен находиться в том же каталоге, что и файл словаря.

Последняя процедура в пакете **DBMS_LOGMNR-DBMS_LOGMNR.END_LOGMNR** Она завершает сеанс LogMiner и очищает представление **V$LOGMNR_CONTENTS.** После вызова **DBMS_LOGMNR.END_LOGMNR** любые попытки обратиться к этому представлению дадут следующий результат:

```
tkyte@TKYTE816> exec dbms_logmnr.end_logmnr;

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select count(*) from v$logmnr_contents;
select count(*) from v$logmnr_contents
```

```
                    *
ERROR at line 1:
ORA-01306: dbms_logmnr.start_logmnr() must be invoked before selecting
from v$logmnr_contents
```

Определение с помощью LogMiner, когда...

Это наиболее типичный вариант использования пакетов LogMiner. Кто-то удалил таблицу. Надо восстановить ее или выяснить, кто это сделал. Другой пример: изменены данные в важной таблице, виновник не признается. Это произошло при отключенной проверке, но база данных работала в режиме архивирования журналов, и все резервные копии доступны. Хотелось бы восстановить данные из резервной копии до момента **непосредственно перед** определенным изменением (например, перед выполнением **DROP** TABLE). Можно восстановить таблицу, прекратив восстановление в нужный момент (чтобы таблица не была удалена снова), экспортировать эту таблицу в восстановленной базе данных, а затем импортировать в текущей. Это позволит восстановить таблицу и оставить в силе остальные изменения.

Для этого необходимо знать либо точное время, либо значение SCN для оператора **DROP TABLE.** Поскольку часы у всех обычно показывают время неточно, а пользователи паникуют, они могут предоставить неверную информацию. Можно загрузить архивные файлы журнала за тот период, когда был выполнен оператор **DROP TABLE,** и найти точное значение SCN, до которого должно выполняться восстановление.

Рассмотрим еще один небольшой пример, показывающий, какие операторы может выдать LogMiner при удалении таблицы. Я использую локально управляемые табличные пространства, так что, если вы используете табличные пространства, управляемые по словарю, у вас может получиться больше SQL-операторов, чем показано далее. Появление дополнительных SQL-операторов в случае управляемых по словарю табличных пространств связано с возвратом экстентов системе и освобождению выделенного таблице пространства. Итак, переходим к удалению таблицы:

```
tkyte@TKYTE816> alter system archive log current;
Systemaltered.
tkyte@TKYTE816> drop table dept;
Table dropped.
tkyte@TKYTE816> alter system archive log current;
Systemaltered.
```

Теперь необходимо найти значение в столбце **SQL_REDO,** представляющее удаление таблицы (оператор **DROP TABLE).** Если помните, LogMiner выдает неточный список выполненных SQL-операторов. Выдаются эквивалентные по действию SQL-операторы. Оператор **DROP TABLE** в результатах работы LogMiner отсутствует: мы увидим только изменения в словаре данных. В частности, нас интересует оператор **DELETE,** примененный к таблице **SYS.OBJ$** — базовой для всех объектов. При удалении таблицы необходимо удалить соответствующую строку из таблицы **SYS.OBJ$.** К счастью, при создании значения столбца **SQL_REDO** для оператора **DELETE** LogMiner включает в

него значения столбцов и идентификатор соответствующей строки. Это можно использовать для поиска оператора **DELETE**, удаляющего строку для **DEPT** из таблицы **OBJ$**, Вот как это делается:

```
tkyte@TKYTE816> declare
  2      l_name v$archived_log.name%type;
  3  begin
  4      select name into l_name
  5        from v$archived_log
  6       where completion_time = (select max(completion_time)
  7                                  from v$archived_log) ;
  8
  9      sys.dbms_logmnr.add_logfile(l_name, sys.dbms_logmnr.NEW);
 10  end;
 11      /
 12

PL/SQL procedure successfully completed.

tkyte@TKYTE816> begin
  2     sys.dbms_logmnr.start_logmnr
  3     (dictFileName => 'c:\temp\miner_dictionary.dat',
  4      options => sys.dbms_logmnr.USE_COLMAP);
  5  end;
  6  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select scn, sql_redo
  2    from v$logmnr_contents
  3   where sql_redo like 'delete from SYS.OBJ$ %''DEPT''%'
  4   /

SCN SQL_REDO

    6442991097246 delete from SYS.OBJ$ where OBJ# = 30553
                  and DATAOBJ# = 30553 and OWNER# = 337 an
                  d NAME = 'DEPT' and NAMESPACE = 1 and SU
                  BNAME IS NULL and TYPE# = 2 and CTIME =
                  TO_DATE('29-APR-2001 12:32:11', 'DD-MON-
                  YYYY HH24:MI:SS') and MTIME = TO_DATE('2
                  9-APR-2001 12:32:11', 'DD-MON-YYYY HH24:
                  MI:SS') and STIME = TO_DATE('29-APR-2001
                   12:32:11', 'DE-MON-YYYY HH24:MI:SS') an
                  d STATUS = 1 and REMOTEOWNER IS NULL and
                   LINKNAME IS NULL and FLAGS = 0 and OID$
                   IS NULL and SPARE1 = 6 and ROWID = 'AAA
                  AASAABAAAFz3AAZ';
```

Вот и все, что нужно. Теперь, получив значение SCN, 6442991097246, можно выполнить восстановление до соответствующего момента, экспортировать таблицу и восстановить ее в системе. Восстановить ее можно в состоянии, непосредственно предшествующем удалению.

Использование области PGA

Для выполнения своих функций пакеты LogMiner используют память в области PGA. Как я уже говорил, это означает, что пакет **DBMS_LOGMNR** нельзя использовать при подключении в режиме MTS. Мы не рассматривали, какой объем памяти в области PGA фактически может использовать LogMiner.

Все файлы журнала в моей системе имеют размер 100 Мбайт. Я загрузил два из них для анализа, определив размер используемой области PGA до и после этого:

```
tkyte@TKYTE816> select a.name, b.value
  2    from v$statname a, v$mystat b
  3   where a.statistic# = b.statistic#
  4     and lower(a.name) like '%pga%'
  5  /

NAME                                VALUE

session pga memory                 454768
session pga memory max             454768

tkyte@TKYTE816> declare
  2        l_name varchar2(255) default
  3             'C:\oracle\ORADATA\tkyte816\archive\TKYTE816T001S012';
  4  begin
  5       for i in 49 .. 50
  6     loop
  7         sys.dbms_logmnr.add_logfile(l_name || i || '.ARC');
  8     end loop;
  9
 10    sys.dbms_logmnr.start_logmnr
 11    (dictFileName => 'c:\temp\miner_dictionary.dat',
 12     options => sys.dbms_logmnr.USE_COLMAP);
 13 end;
 14  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select a.name, b.value
  2    from v$statname a, v$mystat b
  3   where a.statistic# = b.statistic#
  4     and lower(a.name) like '%pga%'
  5  /

NAME                                VALUE

session pga memory                11748180
session pga memory max            11748180
```

Итак, 200 Мбайт архивных журналов повторного выполнения сейчас занимают около 11,5 Мбайт в области PGA. Это означает, что либо большая часть архивного журнала повторного выполнения ничего полезного не содержит, либо сервер Oracle кэширует в памяти не весь файл журнала повторного выполнения. Действительно, сервер Oracle на самом деле не кэширует весь файл журнала повторного выполнения в оперативной па-

мяти. Он подчитывает данные с диска по мере надобности. В оперативной памяти кэшируется только часть информации.

Если выполнить запрос к представлению **V$LOGMNR_CONTENTS** и после этого определить объем используемой памяти в области PGA, мы увидим, что по мере обращения к данным объем используемой памяти растет:

```
tkyte@TKYTE816> create table tmp_logmnr_contents unrecoverable
  2  as
  3  select * from v$logmnr_contents
  4  /

Table created.

tkyte@TKYTE816>select a.name, b.value
  2    from v$statname a, v$mystat b
  3   where a.statistic# = b.statistic#
  4     and lower(a.name) like '%pga%'
  5  /

NAME                                VALUE

session pga memory                 19965696
session pga memory max             19965696
```

Как видите, теперь сеансу надо почти 20 Мбайт памяти в области PGA.

Ограничения пакетов LogMiner

Пакеты LogMiner имеют ряд ограничений, о которых вы должны знать. Ограничения связаны с использованием объектных типов Oracle и переносом строк.

Объектные типы Oracle

Объектные типы лишь частично поддерживаются средствами LogMiner. Пакеты LogMiner не могут восстановить SQL-операторы, обычно используемые для доступа к данным объектных типов, и поддерживают не все объектные типы. Ограничения в этой области лучше всего продемонстрировать на примере. Начнем с небольшой схемы, в которой есть данные таких популярных объектных типов, как VARRAY, и вложенные таблицы:

```
tkyte@TKYTE816> create or replace type myScalarType
  2  as object
  3  (x int, y date, z varchar2(25));
  4  /

Type created.

tkyte@TKYTE816>create or replace type myArrayType
  2  as varray(25) of myScalarType
  3  /

Type created.

tkyte@TKYTE816>create or replace type myTableType
  2  as table of myScalarType
```

```
  3  /

Type created.

tkyte@TKYTE816> drop table t;

Table dropped.

tkyte@TKYTE816> create table t (a int, b myArrayType, c myTableType)
  2  nested table c store as c_tbl
  3  /

Table created.

tkyte@TKYTE816> begin
  2      sys.dbms_logmnr_d.build('miner_dictionary.dat',
  3                              'c:\temp');
  4  end;
  5  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> alter system switch logfile;

System altered.

tkyte@TKYTE816> insert into t values (1,
  2                 myArrayType(myScalarType(2, sysdate, 'hello')),
  3                  myTableType(myScalarType(3, sysdate+1, 'GoodBye'))
  4                                  );

1 row created.

tkyte@TKYTE816> alter system switch logfile;

System altered.
```

Итак, в представленном выше примере мы создали ряд объектных типов, добавили таблицу, использующую эти типы, снова экспортировали словарь данных, а затем выполнили один оператор ЯМД для этой таблицы. Теперь посмотрим, что скажут о выполненных действиях средства LogMiner:

```
tkyte@TKYTE816> begin
  2      sys.dbms_logmnr.add_logfile('C:\oracle\rdbms\ARC00028.001',
  3                                  dbms_logmnr.NEW);
  4  end;
  5  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> begin
  2      sys.dbms_logmnr.start_logmnr
  3      (dictFileName=> 'c:\temp\miner_dictionary.dat');
  4  end;
  5  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select scn, sql_redo, sql_undo
  2    from v$logmnr_contents
  3  /
```

```
       SCN SQL_REDO                    SQL_UNDO

    824288
    824288
    824288
    824288 set transaction read
           write;

    824288 insert into                 delete from
           TKYTE.C_TBL(NESTED_T        TKYTE.C_TBL where
           ABLE_ID,X,Y,Z)              NESTED_TABLE_ID =
           values                      HEXTORAW('252cb5fad8
           (HEXTORAW('252cb5fad        784e2ca93eb432c2d35b
           8784e2ca93eb432c2d35        7c') and X = 3 and Y
           b7c'),3,TO_DATE('23-        =
           JAN-2001 16:21:44',         TO_DATE('23-JAN-2001
           'DD-MON-YYYY                16:21:44',
           HH24:MI:SS'),'GoodBy        'DD-MON-YYYY
           e');                        HH24:MI:SS') and Z =
                                       'GoodBye' and ROWID

                                       'AAAFaqAADAAAAGzAAA'


    824288
    824288
    824288
    824288 insert into                 delete from TKYTE.T
           TKYTE.T(A,B,SYS_NC00        where A = 1 and B =
           00300004$) values           Unsupported Type and
           (1,Unsupported              SYS_NC0000300004$ =
           Type,HEXTORAW('252cb        HEXTORAW('252cb5fad8
           5fad8784e2ca93eb432c        784e2ca93eb432c2d35b
           2d35b7c'));                 7c') and ROWID =
                                       'AAAFapAADAAAARjAAA'
                                       ;

    824288

10 rows selected.
```

Как видите, один исходный оператор **INSERT**:

```
tkyte@TKYTE816> insert into t values (1,
  2                myArrayType(myScalarType(2, sysdate, 'hello')),
  3                 myTableType(myScalarType(3, sysdate+1, 'GoodBye'))
  4                                    );
1 row created.
```

был преобразован в два оператора **INSERT:** один — для подчиненной (вложенной) таблицы, другой — для главной таблицы T. LogMiner не воспроизвел один оператор **INSERT** - он выдал эквивалентный набор SQL-операторов. Посмотрев на результат внимательнее, можно обнаружить в тексте оператора **INSERT INTO** T конструкцию **Unsupported Type** вместо одного из значений столбцов. Возвращаясь к исходному оператору **INSERT**,

можно выяснить, что не поддерживается столбец типа **VARRAY.** Средства LogMiner не позволяют воспроизвести эту конструкцию.

Это **не делает** пакеты LogMiner полностью бесполезными при работе с объектами. Просто результат нельзя использовать для отмены или повторного выполнения транзакций, поскольку соответствующие SQL-операторы воспроизводятся не полностью. Однако можно использовать результат для анализа тенденций, проверки и т.п. Более интересно, пожалуй, то, что пакеты позволяют увидеть, как сервер Oracle внутренне осуществляет поддержку объектных типов. Например, рассмотрим вставку строки в таблицу T:

```
insert into tkyte.t (a, b, SYS_NC0000300004$) values ...
```

Вполне понятно, что такое A и B. Это столбцы типа **INT** и **MyArrayType (VARRAY).** Однако куда делся столбец C и что за столбец **SYS_NC0000300004$?** Столбец C — это вложенная таблица, а вложенные таблицы хранятся в отдельной, подчиненной таблице. Столбец C не хранится в таблице T; он хранится в отдельной таблице. Столбец **SYS_NC0000300004$** — суррогатный первичный ключ для таблицы T, использующийся как внешний ключ во вложенной таблице **C_TBL.** Если рассмотреть оператор **INSERT** для вставки данных во вложенную таблицу:

```
insert into tkyte.c_tbl(nested_table_id, x, y, z) values ...
```

можно увидеть, что во вложенную таблицу добавлен столбец **NESTED_TABLE_ID,** использующийся для соединения со столбцом **T.SYS_NC0000300004$.** Изучив значение, вставленное в оба эти столбца:

```
HEXTORAW('252cb5fad8784e2ca93eb432c2d35b7c')
```

можно выяснить, что сервер Oracle по умолчанию использует для соединения таблиц **C_TBL** и T сгенерированное системой 16-байтовое значение типа **RAW.** Поэтому анализ действий с помощью LogMiner позволяет понять, как реализованы возможности сервера Oracle. В данном случае мы узнали, что тип вложенной таблицы реализуется как пара таблиц главная/подчиненная с суррогатным ключом в главной таблице и внешним ключом в подчиненной.

Перемещенные или фрагментированные строки

Средства LogMiner в настоящий момент не позволяют работать с *перемещенными строками* (migrated row) или *фрагментированными строками* (chained row). Фрагментированной называют строку, расположенную в нескольких блоках. Перемещенной называют строку, вставленную в один блок, а затем в результате изменения выросшую настолько, что она уже не вмещается в исходном блоке и поэтому перемещена в другой блок. Идентификатор у перемещенной строки остается прежним, а в блоке, куда она первоначально вставлялась, остается указатель на новое местонахождение строки. Перемещенные строки являются специальным случаем фрагментированных строк. Это фрагментированная строка, в первом блоке которой нет данных — все данные находятся во втором блоке.

Чтобы разобраться, как средства LogMiner обрабатывают фрагментированные строки, создадим одну такую строку. Начнем с таблицы, содержащей девять столбцов типа

CHAR(2000). Я использую в базе данных блоки размером 8 Кбайт, так что, если во всех девяти столбцах будут непустые значения, строка будет иметь размер 18000 байт, что слишком много для одного блока. Эта строка будет фрагментирована не менее чем на три блока. Для демонстрации этого используем следующую таблицу:

```
tkyte@TKYTE816>create table t (x int primary key,
  2                  a char(2000),
  3                  b char(2000),
  4                  c char(2000),
  5                  d char(2000),
  6                  e char(2000),
  7                  f char(2000),
  8                  g char(2000),
  9                  h char(2000),
 10                  i char(2000));

Table created.
```

Теперь, чтобы продемонстрировать проблему, я вставлю строку в таблицу **T** со значениями только в столбцах **X** и **A**. Размер этой строки будет составлять чуть больше 2000 байт. Поскольку столбцы **B, C,** D и т.д. пусты, они не будут занимать места. Эта строка поместится в один блок. Затем изменим строку, задав значения для столбцов B, **C, D и E**. Поскольку значения типа **CHAR** всегда дополняются до заданной длины пробелами, размер строки увеличится с немногим более 2000 байт до примерно 10000 байт, в результате чего будет фрагментирована на два блока. Изменим значение всех столбцов строки, увеличив ее размер до 18 Кбайт и вызвав фрагментацию на три блока. Затем загрузим содержимое журнала повторного выполнения с помощью LogMiner и посмотоим. как оно будет обработано:

```
tkyte@TKYTE816> begin
  2   sys.dbms_logmnr_d.build('miner_dictionary.dat',
  3                           'c:\temp');
  4  end;
  5  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816>alter system archive log current;

System altered.

tkyte@TKYTE816> insert into t (x, a) values (1, 'non-chained');

1 row created.

tkyte@TKYTE816> commit;

Commit complete.

tkyte@TKYTE816>update t set a = 'chained row',
  2             b = 'x', c = 'x',
  3              d = 'x', e = 'x'

  4   where x = 1;

1 row updated.

tkyte@TKYTE816> commit;
```

```
Commit complete.

tkyte@TKYTE816> update t set a = 'chained row',
  2             b = 'x', c = 'x',
  3             d = 'x', e = 'x',
  4             f = 'x', g = 'x',
  5             h = 'x', i = 'x'
  6   where x = 1;

1 row updated.

tkyte@TKYTE816> commit;

Commit complete.

tkyte@TKYTE816> alter system archive log current;

System altered.
```

Создав состояние, которое мы хотим проанализировать, можно загружать соответствующий журнал с помощью LogMiner. Не забудьте: после создания таблицы T мы должны **пересоздать** файл словаря данных, или результат нельзя будет проанализировать!

```
tkyte@TKYTE816> declare
  2      l_name v$archived_log.name%type;
  3  begin
  4
  5      select name into l_name
  6        from v$archived_log
  7       where completion_time = (select max(completion_time)
  8                                  fromv$archived_log);
  9
 10     sys.dbms_logmnr.add_logfile(l_name, dbms_logmnr.NEW);
 11  end;
 12  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> begin
  2     sys.dbms_logmnr.start_logmnr
  3     (dictFileName=> 'c:\temp\miner_dictionary.dat');
  4  end;
  5  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select scn, sql_redo, sql_undo
  2    from v$logmnr_contents
  3   where sql_redo is not null or sql_undo is not null
  4  /

            SCN SQL_REDO                       SQL_UNDO

   6442991118354 set transaction read write;
   6442991118354 insert into TKYTE.T(X,A) va delete from TKYTE.T where X
                 lues (1,'non-chained        = 1 and A = 'non-chained
```

```
');                                    ' and ROWID
                                  =  'AAAHdgAAGAAAACKAAA';

6442991118355 commit;
6442991118356 set transaction read write;
6442991118356 Unsupported (Chained Row)  Unsupported (Chained Row)
6442991118356 Unsupported (Chained Row)  Unsupported (Chained Row)
6442991118357 commit;
6442991118358 set transaction read write;
6442991118358 Unsupported (Chained Row)   Unsupported (Chained Row)
6442991118358 Unsupported (Chained Row)   Unsupported (Chained Row)
6442991118358 Unsupported (Chained Row)   Unsupported (Chained Row)
6442991118359 commit;

12 rows selected.
```

Как видите, исходный оператор **INSERT** представлен средствами Log Miner, как ожидалось. А оператор **UPDATE,** фрагментировавший строку, в результатах LogMiner не значится. Вместо этого выдано **Unsupported (Chained Row).** Интересно отметить, что эта конструкция выдана дважды для первого оператора **UPDATE** и трижды — для второго. Пакет LogMiner выдает информацию об изменениях в базе данных по блокам. Если строка находится в двух блоках, в представлении **V$LOGMNR_CONTENTS** будет две записи об изменениях. Если строка фрагментирована на три блока, будет три записи. Поэтому следует учитывать, что средства LogMiner не позволяют полностью воспроизвести SQL-операторы для повторного выполнения или отмены изменений во фрагментированных и перемещенных строках.

Другие ограничения

Кроме рассмотренных выше, средства LogMiner имеют ряд других ограничений. В настоящее время также не поддерживается:

- анализ таблиц, организованных по индексу;
- анализ таблиц и индексов кластеров.

Представление V$LOGMNR_CONTENTS

Представление **V$LOGMNR_CONTENTS** содержит по одной строке для каждого логического изменения в базе данных, выбранного из обработанных файлов журнала повторного выполнения. Мы уже неоднократно использовали это представление, но обращались лишь к отдельным его столбцам. В следующей таблице, созданной на основе документации Oracle, описаны все столбцы этого представления, включая доступную в них информацию:

Столбец	*Описание*
SCN	Номер системного изменения для транзакции, выполнившей изменение.

Столбец	*Описание*
TIMESTAMP	Дата генерации записи повторного выполнения. Временные отметки не могут влиять на упорядочение записей повторного выполнения Поскольку значение **SCN** присваивается при фиксации, только **SCN** можно использовать для упорядочения записей повторного выполнения. Упорядочение по столбцу **TIMESTAMP** в многопользовательской системе даст неверный порядок
THREAD#	Идентифицирует поток, сгенерировавший запись повторного выполнения
LOG_ID	Идентифицирует файл журнала в представлении **V$LOGMNR_FILES**, содержащий данную запись повторного выполнения. Этот столбец является внешним ключом для представления **V$LOGMNR_FILES**
XIDUSN	Номер сегмента отмены (**U**ndo **S**egment **N**umber - **USN**) идентификатора транзакции (Transaction **ID** - XID). Идентификатор транзакции создается по значениям столбцов **XIDUSN, XIDSLOT и XIDSQN** и используется для определения транзакции, выполнившей изменение. Вместе взятые, эти три поля однозначно идентифицируют транзакцию
XIDSLOT	Номер слота идентификатора транзакции. Задает номер записи в таблице транзакций
XIDSQN	Порядковый номер идентификатора транзакции
RBASQN	Однозначно определяет журнал, содержавший рассматриваемую запись повторного выполнения. Значение RBA (Redo Block Address - адрес блока повторного выполнения) состоит из значений столбцов **RBASQN, RBABLK и RBABYTE**
RBABLK	Номер блока в файле журнала
RBABYTE	Смещение от начала блока, задаваемого предыдущим столбцом
UBAFIL	Номер файла UBA (Undo Block Address - адрес блока отмены), идентифицирующий файл, содержащий соответствующий блок отмены. Значение UBA создается на основе значений столбцов **UBAFIL, UBABLK, UBASQN и UBAREC** и используется для идентификации данных отмены, сгенерированных в процессе изменения
UBABLK	Номер блока UBA
UBAREC	Индекс записи UBA
UBASQN	Порядковый номер блока отмены UBA
ABS_FILE#	Абсолютный номер файла блока данных. Значение **ABS_FILE#** вместе со значениями **REL_FILE#, DATA_BLOCK#, DATA_OBJ#** и **DATA_DOBJ** идентифицирует блок, измененный транзакцией
REL_FILE#	Относительный номер файла блока данных. Задается относительно табличного пространства, в котором создан объект
DATA_BLOCK#	Номер блока данных

Столбец	*Описание*
DATA_OBJ#	Номер объекта блока данных
DATA_DOBJ#	Номер объекта блока данных, идентифицирующий объект в табличном пространстве
SEG_OWNER	Имя пользователя, которому принадлежит объект
SEG_NAME	Имя структуры, которой был выделен сегмент (другими словами, имя таблицы, имя кластера и т.п.). Имя сегмента для фрагментированных таблиц состоит из двух частей: после имени таблицы через запятую выдается имя фрагмента (например, **(ИмяТаблицы.ИмяФрагмента))**
SEG_TYPE	Тип сегмента в виде числа
SEG_TYPE_NAME	Тип сегмента в виде строки (другими словами, **TABLE, INDEX** и т.п.) В первой версии будет поддерживаться единственный тип сегмента, **TABLE.** Сегменты остальных типов будут обозначаться как **UNSUPPORTED**
TABLE_SPACE_NAME	Имя табличного пространства
ROW_ID	Идентификатор строки
SESSION#	Идентификатор сеанса, сгенерировавшего данные повторного выполнения. Если номер сеанса из журнала повторного выполнения получить не удается, выдается пустое значение
SERIAL#	Порядковый номер сеанса, сгенерировавшего данные повторного выполнения. Значения **SESSION#** и **SERIAL#** позволяют однозначно определить соответствующий сеанс сервера Oracle. Если порядковый номер сеанса из журнала повторного выполнения получить не удается, выдается пустое значение
USERNAME	Имя пользователя, который выполнил действие, сгенерировавшее запись повторного выполнения. Вместо имени пользователя всегда будет выдаваться пустое значение, если только не включена опция проверки при архивировании. Проверка включается с помощью параметра инициализации **TRANSACTION_AUDITING**
SESSION_INFO	Строка, содержащая регистрационное имя пользователя, информацию о клиенте, имя пользователя операционной системы, имя машины, терминал операционной системы и имя программы в операционной системе
ROLLBACK	Значением 1 (ИСТИНА) обозначаются действия и SQL-операторы, сгенерированные в результате запроса отката. В противном случае в этом столбце содержится значение 0 (ЛОЖЬ)
OPERATION	Тип операции SQL. Будут выдаваться только значения **INSERT, DELETE, UPDATE, COMMIT** и **BEGIN_TRANSACTION.** Все остальные операции будут представлены как **UNSUPPORTED** или **INTERNAL OPERATION**

Столбец	*Описание*
SQL_REDO, SQL_UNDO	Столбцы **SQL_REDO** и **SQL_UNDO** содержат SQL-подобные операторы, представляющие логические операции повторного выполнения и отмены, построенные на основе одной или нескольких записей архивного журнала повторного выполнения. Пустое значение показывает, что для этой записи повторного выполнения нельзя сгенерировать допустимый SQL-оператор. Некоторые записи повторного выполнения нельзя преобразовать в SQL-операторы. В этом случае в столбцах **SQL_REDO** и **SQL UNDO** будут содержаться пустые значения, а в столбце **STATUS** - строка **UNSUPPORTED**
RS_ID	Значение **RS_ID** (Record Set ID - идентификатор набора записей) однозначно определяет набор записей, использованных для генерации SQL-оператора (набор может состоять из одной записи). Это значение можно использовать для выявления ситуаций, когда несколько записей генерируют один SQL-оператор. Во всех записях соответствующего набора будет одинаковое значение **RS_ID.** SQL-оператор будет выдан только в последней строке набора. Столбцы **SQL_REDO** и **SQL_UNDO** во всех остальных строках набора будут пустыми. Учтите, что пара значений **RS_ID**/**SSN** уникально идентифицирует сгенерированный SQL-оператор (см. описание столбца **SSN)**
SSN	Значение **SSN** (SQL Sequence Number - порядковый номер SQL-оператора) можно использовать для идентификации нескольких строк с допустимыми операторами **SQL_REDO,** сгенерированных по одной записи повторного выполнения (при множественных вставках или непосредственной загрузке). Все такие строки будут иметь одинаковое значение **RS_ID,** но уникальные значения **SSN.** Значение **SSN** увеличивается, начиная с 1 для каждого нового значения **RS_ID**
CSF	Значение 1 (ИСТИНА) в столбце **CSF** (Continuation SQL Flag - флаг продолжающегося SQL-оператора) показывает, что сгенерированный средствами LogMiner оператор **REDO_SQL** или **UNDO_SQL** длиннее, чем максимальный размер данных типа **VARCHAR2** (в настоящее время - 4000 символов). SQL-операторы, длина которых превышает это ограничение, будут занимать несколько строк. Остальная часть SQL-оператора будет содержаться в следующей строке. Пара значений **RS_ID, SSN** у всех продолжающихся строк, которые соответствуют одному SQL-оператору, будет одинаковой. В последней строке значение в столбце **CSF** будет равно 0 (ЛОЖЬ), что обозначает завершение SQL-оператора
STATUS	Показывает состояние преобразования. Пустое значение означает успешное преобразование, а значение **UNSUPPORTED** означает, что эта версия пакетов LogMiner не поддерживает преобразование в SQL-операторы. Значение **READ_FAILURE** свидетельствует о внутреннем сбое операционной системы при попытке прочитать данные из файла журнала. Значение **TRANSLATION_ERROR** показывает, что LogMiner не смог полностью выполнить преобразование (это может быть связано с повреждением журнала или устаревшим файлом словаря данных)

Столбец	*Описание*
PH1_NAME	Имя столбца-заместителя (Placeholder column name). Столбцы-заместители - это общие столбцы, с которыми можно связать любой столбец таблицы базы данных с помощью файла сопоставления LogMiner
PH1_REDO	Значение повторного выполнения для столбца-заместителя
PH1_UNDO	Значение отмены для столбца-заместителя
PH2_NAME	Имя столбца-заместителя
PH2_REDO	Значение повторного выполнения для столбца-заместителя
PH2_UNDO	Значение отмены для столбца-заместителя
PH3_NAME	Имя столбца-заместителя
PH3_REDO	Значение повторного выполнения для столбца-заместителя
PH3_UNDO	Значение отмены для столбца-заместителя
PH4_NAME	Имя столбца-заместителя
PH4_REDO	Значение повторного выполнения для столбца-заместителя
PH4_UNDO	Значение отмены для столбца-заместителя
PHS_NAME	Имя столбца-заместителя
PH5_REDO	Значение повторного выполнения для столбца-заместителя
PH5_UNDO	Значение отмены для столбца-заместителя

Резюме

Средства LogMiner используются не каждый день. Я не могу вспомнить ни одного приложения, которому бы они понадобились по ходу работы. Однако они позволяют определить, что происходило в базе данных, и с этой задачей справляются прекрасно. Вы видели, как пакеты LogMiner помогают при поиске "кто и когда это сделал" постфактум — именно для этого средства LogMiner используются. Или программа с ошибкой выполняется неправильно, или привилегированный пользователь сделал не то, что нужно (и не признается в содеянном). Если не была включена проверка, нет другого способа вернуться в прошлое и разобраться в том, что произошло. Средства LogMiner можно использовать и для отмены ошибочной транзакции, если удастся получить операторы SQL длям отмены и повторного выполнения. В общем случае LogMiner — хорошее средство для "скрытой" работы. Процедуры этих пакетов не попадут в список десяти наиболее часто используемых, но иногда без них не обойтись.

Пакет DBMS_OBFUSCATION_TOOLKIT

В этом разделе рассматривается шифрование данных. Мы обсудим использование стандартного пакета **DBMS_OBFUSCATION_TOOLKIT** в версиях Oracle 8.1.6 и 8.1.7, а также другую реализацию (оболочку), которую можно создать на базе этого пакета для расширения его функциональных возможностей. Мы затронем ряд проблем, связанных с использованием этого пакета, и очень важный аспект управления ключами.

Пакеты для поддержки шифрования в базе данных появились в Oracle 8.1.6. Они были расширены в версии Oracle 8.1.7 и включают поддержку ключей шифрования большего размера и алгоритма хеширования MD5. В Oracle 8.1.6 поддерживается одинарный алгоритм шифрования **DES** (**D**ata **E**ncryption **S**tandard) с 56-байтовым ключом. В Oracle 8.1.7 поддерживаются одинарное и тройное шифрование **DES** с использованием 56-, 112- и 168-битовых ключей. Алгоритм **DES** реализует шифрование с симметричным ключом. Это означает, что для шифрования и дешифрования данных используется один и тот же ключ. Алгоритм **DES** шифрует данные 64-битовыми (8-байтовыми) блоками с помощью 56-битового ключа. Ниже мы рассмотрим, как размер блока, 8 байт, должен учитываться пользователями подпрограмм шифрования. Алгоритм DES игнорирует 8 бит переданного 64-битового ключа. Однако разработчики должны передавать 64-битовый (8-байтовый) ключ. Тройной DES (3DES) гораздо устойчивей к взлому по сравнению с DES. Зашифрованные данные сложно расшифровать с помощью полного перебора. Потребуется 2**112 попыток при использовании двойного (с 16-байтовым ключом) алгоритма 3DES или 2**168 попыток при использовании тройного (с 24-байтовым ключом) алгоритма 3DES и всего лишь 2**56 попыток для обычного алгоритма **DES**.

Как сказано в заключительной части рабочего документа **rfcl321** (полный текст рабочего документа **rfcl321** можно найти на сайте **http://www.ietf.org/rfc.html),** новый алгоритм MD5:

> ... принимает сообщение произвольной длины и выдает 128-битовый "отпечаток" или резюме сообщения ("message digest"). Предполагается, что задача создания двух сообщений с одинаковым резюме или сообщения с заданным резюме эффективно неразрешима. Алгоритм **MD5** предназначен для приложений, использующих цифровую подпись, в которых большой файл необходимо безопасно "сжать", прежде чем шифровать приватным (секретным) ключом в системе шифрования с открытым ключом, например, **RSA**

По сути, алгоритм **MD5** — это способ проверки целостности данных, намного более надежный, чем контрольные суммы и другие популярные методы.

Для выполнения представленных ниже примеров использования алгоритмов DES3 и MD5 необходим доступ к серверу Oracle 8.1.7. Примеры, в которых используется алгоритм **DES**, работают во всех версиях, начиная с Oracle 8.1.6.

При использовании подпрограмм шифрования и резюмирования по алгоритму **MD5** пакета **DBMS_OBFUSCATION_TOOLKIT** должны выполняться следующие требования, что несколько затрудняет их применение.

- Длина шифруемых данных должна быть кратна 8. 9-байтовое поле типа VARCHAR2, например, надо будет дополнять до 16 байт. Попытка зашифровать или расшифровать фрагмент данных, длина которого не кратна 8, завершится выдачей сообщения об ошибке.
- Ключ, используемый для шифрования данных, должен иметь длину 8 байт для процедуры **DESEncrypt** и 16 или 24 байт для процедур **DES3Decrypt**.
- В зависимости от используемого вида шифрования необходимо вызывать разные подпрограммы. Например, если используется 56-битовое шифрование, вызываются подпрограммы **DESENCRYPT** и **DESDECRYPT**, если 112/168-битовое - подпрограммы **DES3ENCRYPT** и **DES3DECRYPT**. Лично я предпочитаю использовать один набор подпрограмм для всех трех вариантов.
- Подпрограммы шифрования в версии Oracle 8.1.6 являются процедурами, поэтому их нельзя использовать в SQL-операторах (процедуры нельзя вызывать в SQL-операторах непосредственно).
- Стандартные подпрограммы шифрования позволяют непосредственно шифровать данные объемом до 32 Кбайт. Они не обеспечивают шифрование/дешифрование больших объектов.
- Среди подпрограмм шифрования в версии Oracle 8.1.7 есть функции. Однако эти функции перегружены так (см. представленные далее примеры), что использовать их в SQL-операторах тоже нельзя.
- Подпрограммы, реализующие алгоритм **MD5**, перегружены аналогично и тоже не могут использоваться в SQL-операторах.

По моему опыту, первое требование, связанное с необходимостью использовать данные, длина которых кратна 8, в приложениях выполнить труднее всего. Мне не хотелось бы заниматься тем, чтобы длина шифруемых данных, например зарплаты или других конфиденциальных данных, была кратна 8 байтам. К счастью, можно легко реализовать пакет-оболочку, обеспечивающий шифрование без учета этого и большин-

ства других требований. А вот обеспечение длины ключа 8, 16 или 24 байт должен взять на себя разработчик.

Я собираюсь представить здесь пакет-оболочку, который будет работать в версиях 8.1.6 и выше, поддерживать все возможности шифрования и обеспечивать:

- возможность вызова функций в SQL-операторах;
- использование одной и той же функции независимо от длины ключа;
- возможность шифрования/дешифрования больших объектов в PL/SQL и в SQL-операторах;
- успешную установку независимо от используемой версии сервера (8.1.6 или 8.1.7) — другими словами, независимость от наличия процедур **DES3Encrypt/Decrypt** и поддержки алгоритма **MD5**.

Пакет-оболочка

Начнем со спецификации пакета. Зададим функциональный интерфейс, обеспечивающий шифрование и дешифрование данных типа **VARCHAR, RAW, BLOB** и **CLOB.** Используемый алгоритм (DES или 3DES с 16- или 24-байтовым ключом) будет выбираться в зависимости от длины ключа. В нашем интерфейсе длина ключа будет задавать алгоритм.

Интерфейс устроен так, что ключ можно передавать при каждом вызове или установить на уровне пакета с помощью подпрограммы **SETKEY**. Преимущество использования подпрограммы **SETKEY** связано с тем, что проверка длины ключа и определение соответствующего алгоритма требует определенных ресурсов. Если ключ устанавливается один раз, а функции шифрования вызываются многократно, можно избежать повторного выполнения одних и тех же действий. Еще одна особенность при задании ключа состоит в том, что при работе с данными типа **RAW** или **BLOB** надо использовать ключ типа **RAW.** Если в качестве ключа для данных типа **RAW/BLOB** желательно использовать данные типа **VARCHAR2,** необходимо привести ключ к типу **RAW** с помощью средств пакета **UTL_RAW,** который рассматривается далее в этом приложении. С другой стороны, при работе с данными типа **VARCHAR2** и **CLOB,** ключ должен быть типа **VARCHAR2.**

Помимо организации дополнительного уровня интерфейса к средствам шифрования этот пакет обеспечивает доступ к подпрограммам **CHECKSUM** с алгоритмом MD5, если они установлены (используется сервер версии 8.1.7 и выше).

Этот пакет-оболочка добавляет несколько возможных сообщений об ошибках к тем, что имеются в пакете **DBMS_OBFUSCATION_TOOLKIT** и описаны в документации (эти сообщения наш пакет просто передает вызывающему). В процессе работы с сервером версии 8.1.6 выдаются такие, не известные ранее сообщения:

- `PLS-00302: component 'MD5' must be declared`
- `PLS-00302: component 'DES3ENCRYPT' must be declared`
- `PLS-00302: component 'THREEKEYMODE' must be declared`

Эти сообщения об ошибках генерируются при попытке использовать средства версии 8.1.7, шифрование по алгоритму DES3 или хеширование с помощью алгоритма MD5, на сервере версии 8.1.6.

Вот предполагаемая спецификация пакета-оболочки. Описание процедур и функций представлено после кода:

```
create or replace package crypt_pkg
as
function encryptString(p_data in varchar2,
                       p_key in varchar2 default NULL) return varchar2;
function decryptString(p_data in varchar2,
                       p_key in varchar2 default NULL) return varchar2;

function encryptRaw(p_data in raw, p_key in raw default NULL) return raw;
function decryptRaw(p_data in raw, p_key in raw default NULL) return raw;

function encryptLob(p_data in clob,
                    p_key in varchar2 default NULL) return clob;
function encryptLob(p_data in blob,
                    p_key in raw default NULL) return blob;
function decryptLob(p_data in clob,
                    p_key in varchar2 default NULL) return clob;
function decryptLob(p_data in blob,
                    p_key in raw default NULL) return blob;

subtype checksum_str is varchar2(16);
subtype checksum_raw is raw(16);

function md5str(p_data in varchar2) return checksum_str;
function md5raw(p_data in raw) return checksum_raw;
function md51ob(p_data in clob) return checksum_str;
function md51ob(p_data in blob) return checksuro_raw;
procedure setKey(p_key in varchar2);

end;
/
```

Функции **ENCRYPTSTRING** и **DECRYPTSTRING** используются для шифрования/дешифрования любых данных типа **STRING**, **DATE** или **NUMBER** длинной до 32 Кбайт. Максимальный размер PL/SQL-переменной — 32 Кбайт, что существенно превышает максимальный размер строки, сохраняемой в базе данных, — 4000 байт. Эти функции можно вызывать непосредственно из SQL-операторов, так что, можно шифровать данные в базе с помощью операторов **INSERT** или **UPDATE** и выбирать уже расшифрованные данные с помощью оператора **SELECT**. Параметр **KEY** — необязательный. Если ключ установлен с помощью процедуры **SETKEY**, передавать его при каждом вызове необязательно.

Имеются также функции **ENCRYPTRAW** и **DECRYPTRAW**. Они реализуют для данных типа **RAW** те же возможности, что и функции для данных типа **VARCHAR2**. Я намеренно избегаю перегрузки функций шифрования/дешифрования для данных типа RAW и **VARCHAR2**, давая им разные имена. Это делается для того, чтобы избежать следующей проблемы:

```
tkyte8TKYTE816> create or replace package overloaded
  2  as
  3        function foo(x in varchar2) return number;
  4        function foo(x in raw) return number;
  5  end;
  6  /

Package created.

tkyte@TKYTE816>select overloaded.foo('hello') from dual;
select overloaded.foo('hello') from dual
       *
ERROR at line 1:
ORA-06553: PLS-307: too many declarations of 'P00' match this call


tkyte@TKYTE816>select overloaded.foo(hextoraw('aa')) from dual;
select overloaded.foo( hextoraw('aa')) from dual
       *
ERROR at line 1:
ORA-06553: PLS-307: too many declarations of 'F00' match this call
```

Сервер не различает типы данных **RAW** и **VARCHAR2** в сигнатурах перегруженных функций. Вызвать эти функции в SQL-операторах невозможно. Даже если использовать другие имена параметров функций (как в пакете **DBMS_OBFUSCATION_TOOLKIT), нельзя вызвать** эти функции из SQL-операторов, поскольку ключевая передача параметров в языке SQL не поддерживается. Единственно возможное решение — использовать функции с уникальными именами для идентификации необходимого типа данных параметра.

Функции **ENCRYPTLOB** и **DECRYPTLOB** — это перегруженные функции, предназначенные для работы с данными типа **CLOB** или **BLOB.** Сервер Oracle позволяет различать эти типы в сигнатурах, и этим можно воспользоваться. Поскольку нельзя шифровать более 32 Кбайт с помощью подпрограмм пакета **DBMS_OBFUSCATION_TOOLKIT,** эти функции будут использовать алгоритм шифрования больших объектов частями по 32 Кбайт. Полученный большой объект будет представлять собой набор зашифрованных фрагментов данных размером по 32 Кбайт. Оболочка дешифрования, которую мы реализуем, учитывает упаковку данных подпрограммами шифрования больших объектов, дешифруя фрагменты и собирая из них исходный большой объект.

Затем в пакете идут подпрограммы для вычисления контрольных сумм по алгоритму MD5. Чтобы точнее задать тип возвращаемых ими значений, я создал подтипы:

```
subtype checksum_str is varchar2(16);
subtype checksum_raw is raw(16);
```

и указал, что эти функции возвращают именно эти типы. Пользователи пакета могут объявлять переменные этих типов:

```
tkyte@TKYTE816> declare
  2    checksum_variable  crypt_pkg.checksum_str;
  3  begin
  4   null;
  5  end;
```

```
6  /

PL/SQL procedure successfully completed.
```

Это позволяет не гадать, каким будет размер возвращаемых контрольных сумм. Для данных типа **VARCHAR2** предлагается четыре функции **CHECKSUM** (включая типы **DATE** и **NUMBER**, для которых выполняется неявное преобразование), **RAW**, **CLOB** и **BLOB**. Следует помнить, что контрольная сумма по алгоритму **MD5** будет вычисляться только по первым 32 Кбайтам данных типа **CLOB** или **BLOB**, поскольку с переменными большего размера в языке PL/SQL работать нельзя.

Представленная далее реализация пакета не только дает более удобные в использовании средства шифрования, но и демонстрирует несколько полезных приемов. Во-первых, она показывает, как легко создать свою оболочку со специализированным интерфейсом для стандартных пакетов базы данных. В данном случае мы обходим ряд очевидных ограничений пакета **DBMS_OBFUSCATION_TOOLKIT**. Во-вторых, показан один из методов разработки пакета, защищенного от будущих изменений в реализации стандартных пакетов. Хотелось бы создать один пакет-оболочку, который будет работать в версиях 8.1.6 и 8.1.7 и обеспечивать при этом полный доступ к возможностям версии 8.1.7. Если бы для доступа к подпрограммам **DESENCRYPT**, **DES3DECRYPT** и средствам поддержки алгоритма MD5 использовался статический SQL, пришлось бы создавать отдельную версию пакета для версии 8.1.6 сервера, поскольку функций для алгоритмов **MD5** и **DES3** в версии 8.1.6 нет. Использованный в реализации динамический вызов позволяет создать пакет, который можно использовать в обеих версиях сервера. При этом также сокращается объем кода, который необходимо написать.

Вот реализация пакета **CRYPT_PKG** с описанием выполняемых действий.

```
create or replace package body crypt_pkg
as
— глобальные переменные пакета
g_charkey          varchar2(48);
g_stringFunction   varchar2(1);
g_rawFunction      varchar2(1);
g_stringWhich      varchar2(75);
g_rawWhich         varchar2(75);
g_chunkSize        CONSTANT number default 32000;
```

Пакет начинается с объявления глобальных переменных и констант.

- **G_CHARKEY**. В ней хранится ключ типа RAW или VARCHAR2 для использования подпрограммами шифрования. Ее длина — 48 байт, чтобы можно было хранить 24-байтовый ключ типа RAW (при этом длина удваивается из-за преобразования в шестнадцатеричный вид данных типа RAW при записи в переменную типа VARCHAR2).
- **G_STRINGFUNCTION** и **G_RAWFUNCTION**. Содержит после вызова **SETKEY** пустое значение либо строку '3'. Мы будем динамически добавлять эту строку к имени подпрограммы при выполнении, чтобы вызывать **DESENCRYPT** либо **DES3ENCRYPT**, в зависимости от размера ключа. Другими словами, она используется для формирования имени вызываемой функции.

- **G_STRINGWHICH и G_RAWWHICH.** Используется только для функций **DES3ENCRYPT/DES3DECRYPT.** Добавляет четвертый необязательный параметр, требующий использования тройного ключа, когда алгоритм 3DES работает в этом режиме. Тогда как предыдущая строковая переменная определяет, какую из функций вызывать: **DESENCRYPT** или **DES3ENCRYPT,** эта переменная задает, какое значение должно быть передано для размера ключа: для двойного или тройного.
- G_CHUNKSIZE. Константа, задающая размер фрагмента шифруемого/дешифруемого большого объекта. Она также задает максимальный размер данных, посылаемых функциям вычисления контрольной суммы по алгоритму **MD5** при работе с большими объектами. **Важно,** чтобы это значение было кратно 8, — это предполагается в представленной далее реализации.

Далее имеется шесть небольших служебных подпрограмм. Они являются вспомогательными и используются другими подпрограммами пакета:

```
function padstr(p_str in varchar2) return varchar2
as
    l_len number default length(p_str);
begin
    return to_char(l_len,'fm00000009') ||
           rpad(p_str,  (trunc(l_len/8)+sign(mod(l_len,8)))*8,  chr(0));
end;

function padraw(p_raw in raw) return raw
as
    l_len number default utl_raw.length(p_raw);
begin
    return
utl_raw.concat(utl_raw.cast_to_raw(to_char(l_len,'fm00000009')),
                          p_raw,
                          utl_raw.cast_to_raw(rpad(chr(0),
                          (8-mod(l_len,8))*sign(mod(l_len,8)),
                          chr(0))));
end;
```

При описании алгоритма шифрования DES было сказано, что **DES шифрует данные блоками по 64 бита (8 байт).** Значит, пакет **DBMS_OBFUSCATION_TOOLKIT** работает **только** с данными, длина которых кратна 8 байтам. Если шифруется строка длиной 7 байт, ее надо дополнить до 8. 9-байтовую строку необходимо дополнить до 16 байт. Две представленных выше подпрограммы кодируют и дополняют до нужной длины строки и данные типа **RAW.** Кодирование происходит путем помещения значения исходного размера данных перед данными в строку. Затем строка дополняется двоичными нулями **(CHR(0))** до длины, кратной 8 байтам. Например, строка **Hello World** будет закодирована следующим образом:

```
tkyte@TKYTE816>select length(padstr), padstr, dump(padstr) dump
  2    from
  3  (selectto_char(l_len,'fm00000009')  ||
  4            zpad(p_str,
  5                (trunc(l_len/8)+sign(mod(l_len,8)))*8,
```

```
  6         chr(0))         padstr
  7       from (select length('Hello World') l_len,
  8                    'Hello World' p_str
  9                from dual
 10             )
 11    )
 12  /

LENGTH(PADSTR) PADSTR                        DUMP

            24 00000011Hello World     Typ=1 Len=24: 48,48,48,48,48,4
                                         8,49,49,72,101,108,108,111,32,
                                         87,111,114,108,100,0,0,0,0,0
```

Окончательная длина закодированной строки — 24 байта (**LENGTH(PADSDTR)**), а исходная длина была 11 байт (это видно в первых восьми символах значения **PADSTR**). В столбце **DUMP**, где выданы десятичные значения байтов строки, можно увидеть, что строка завершается пятью двоичными нулями. Нам пришлось добавить 5 байт, чтобы дополнить 11-байтовую строку **Hello World** до длины, кратной 8. Далее идут подпрограммы, "отменяющие" выполненное выше дополнение нулевыми байтами:

```
function unpadstr(p_str in varchar2) return varchar2
is
begin
    return substr(p_str, 9, to_number(substr(p_str,1,8)));
end;

function unpadraw(p_raw in raw) return raw
is
begin
    return utl_raw.substr(p_raw, 9,
        to_number(utl_raw.cast_to_varchar2(utl_raw.substr(p_raw,1,8))));
end;
```

Они достаточно понятны. Предполагается, что в первых восьми байтах строки или данных типа **RAW** находится исходная длина строки, и возвращается соответствующая подстрока закодированных данных.

Осталось две вспомогательные процедуры:

```
procedure wa(p_clob in out clob, p_buffer in varchar2)
is
begin
   dbms_lob.writeappend(p_clob, length(p_buffer), p_buffer);
end;

procedure wa(p_blob in out blob, p_buffer in raw)
is
begin
   dbms_lob.writeappend(p_blob, utl_raw.length(p_buffer), p_buffer);
end;
```

Они упрощают вызов **DBMS_LOB.WRITEAPPEND**, сокращая имя до двух букв (**WA**) и передавая длину записываемого фрагмента буфера — она всегда совпадает с полной длиной буфера.

Теперь рассмотрим первую общедоступную процедуру, **SETKEY:**

```
procedure setKey(p_key in varchar2)
as
begin
    if (g_charkey = p_key OR p_key is NULL) then
        return;
    end if;
    g_charkey := p_key;

    if (length(g_charkey) not in (8, 16, 24, 16, 32, 48))
    then
        raise_application_error(-20001,
                            'Key must be 8, 16, or 24 bytes');
    end if;

    select decode(length(g_charkey),8,'','3'),
           decode(length(g_charkey),8,'',16,'',
               24,',which=>dbms_obfuscation_toolkit.ThreeKeyMode'),
           decode(length(g_charkey),16, '', '3'),
           decode(length(g_charkey),16,'•,32,'',
               48,',which=>dbms_obfuscation_toolkit.ThreeKeyMode')
      into g_stringFunction, g_stringWhich, g_rawFunction, g_rawWhich
      from dual;
end;
```

Процедура используется независимо от того, вызывали вы ее или нет. Остальные общедоступные подпрограммы вызывают процедуру **SETKEY** всегда. Она сравнивает переданный ключ **P_KEY** с тем, что хранится в глобальной переменной **G_CHARKEY.** Если они совпадают или ключ не задан, процедура завершает работу. Если же значение **P_KEY** отличается от значения **G_CHARKEY,** процедура продолжит работу. Сначала она проверит, допустима ли длина ключа и кратна ли 8. Ключ должен быть длиной 8, 16 или 24 байт. Поскольку процедуре могут быть переданы данные типа **RAW,** что вызывает представление каждого байта двухбайтовым шестнадцатиричным кодом, допускается также длина ключа 16, 32 и 48. Такая проверка, однако, не гарантирует, что ключ можно использовать. Например, можно передать четырехбайтовый ключ типа **RAW,** который в процедуре будет иметь длину 8 байт. В этом случае при дальнейшем выполнении подпрограмм пакета **DBMS_OBFUSCATION_TOOLKIT** будет получено сообщение об ошибке.

Оператор **SELECT** с функцией **DECODE** используется для установки значений остальных глобальных переменных. Поскольку мы пока не можем различить типы данных **RAW** и **VARCHAR2,** то устанавливаем значения всем четырем переменным. Главное в этом фрагменте кода то, что если длина ключа — 8 байт (16 байт, если он типа **RAW),** то переменная FUNCTION получит значение пустой строки. Если же длина ключа — 16 или 24 байт (32 или 48 байт для ключа типа **RAW),** в переменную **FUNCTION** записывается строка '3'. Именно это в дальнейшем позволит вызвать подпрограмму **DESENCRYPT** или **DES3Encrypt.** Обратите внимание также на установку значения глобальной переменной **WHICH.** Она используется для передачи необязательного параметра подпрограмме **DES3ENCRYPT.** Если длина ключа — 8 или 16 байт (16 или 32 байта для ключа типа **RAW),** переменная получает значение **Null,** — параметр не передается. Если

длина ключа — 24 байта (48 байт для ключа типа **RAW**), она получает значение **THREEKEYMODE**, требующее от процедур **ENCRYPT/DECRYPT** использовать ключ большего размера.

Теперь мы готовы к рассмотрению функций, выполняющих основные действия:

```
function encryptString(p_data in varchar2,
                       p_key in varchar2 default NULL) return varchar2
as
    l_encrypted long;
begin
    setkey(p_key);
    execute immediate
     'begin
        dbms_obfuscation_toolkit.des' || g_StringFunction || 'encrypt
         (input_string => :1, key_string => :2, encrypted_string => :3' ||
        g_stringWhich || ' ) ;
      end;'
      using IN padstr(p_data), IN g_charkey, IN OUT l_encrypted;

    return l_encrypted;
end;

function encryptRaw(p_data in raw,
                    p_key in raw default NULL) return raw
as
    l_encrypted long raw;
begin
    setkey(p_key);
    execute immediate
     'begin
        dbms_obfuscation_toolkit.des' || g_RawFunction || 'encrypt
         (input => :1, key => :2, encrypted_data => :3' ||
        g_rawWhich  ||  ' ) ;
      end;'
      using IN padraw(p_data), IN hextoraw(g_charkey), IN OUT l_encrypted;

    return l_encrypted;
end;
```

Функции **ENCRYPTSTRING** и **ENCRYPTRAW** действуют одинаково. Они обе *динамически* вызывают процедуру **DESENCRYPT** либо DES3ENCRYPT. Этот динамический вызов не только сокращает объем необходимого кода (поскольку избавляют от оператора **IF THEN ELSE** для статического вызова процедур), но и позволяют устанавливать пакет без изменений в версии 8.1.6 или 8.1.7. Поскольку мы не ссылаемся на подпрограммы пакета **DBMS_OBFUSCATION_TOOLKIT** статически, то сможем скомпилировать функцию в любой версии. Этот прием с динамическим вызовом пригодится в том случае, когда точно не известно, что будет установлено в базе данных. Я использовал его ранее при написании утилит, которые должны были устанавливаться на серверах версии 7.3, 8.0 и 8.1. Со временем в базовых пакетах появляются новые функции, и, если код работает в версии 8.1, хотелось бы их использовать. В версии 7.3 код тоже будет работать; в нем просто нельзя будет использовать новые функциональные возможное-

ти. В нашем случае именно так и происходит. При установке пакета на сервере версии 8.1.7 представленный выше код будет вызывать процедуру **DES3ENCRYPT**. При установке пакета на сервере версии 8.1.6любая попытка вызвать процедуру **DES3ENCRYPT** приведет к ошибке времени выполнения (но не помешает установить пакет). Вызовы процедуры **DESENCRYPT** будут работать так же, как в версии 8.1.6.

Эти функции динамически формируют строку на основе значений глобальных переменных **FUNCTION** и **WHICH**, которые мы установили в процедуре **SETKEY**. Мы либо добавим цифру 3 к имени процедуры, либо нет. Мы добавим необязательный четвертый параметр к вызову процедуры **DES3ENCRYPT**, если необходимо использовать тройной ключ. Затем выполняем строку, посылая данные и ключ для шифрования, и получаем данные в зашифрованном виде. Обратите внимание, как к исходным данным подставляются результаты применения функций **PADSTR** или **PADRAW**. Шифруется закодированная строка, дополненная до соответствующей длины.

Теперь рассмотрим функции, выполняющие обратные действия:

```
function decryptString(p_data in varchar2,
                       p_key in varchar2 default NULL) return varchar2
as
    l_string long;
begin
    setkey(p_key);
    execute immediate
    'begin
       dbms_obfuscation_toolkit.des' || g_StringFunction || 'decrypt
        (input_string => :1, key_string => :2, decrypted_string => :3' ||
       g_stringWhich||' ) ;
     end;'
     using IN p_data, IN g_charkey, IN OUT l_string;

    return unpadstr(l_string);
end;

function decryptRaw(p_data in raw,
                    p_key in raw default NULL) return raw
as
    l_string long raw;
begin
    setkey(p_key);
    execute immediate
    'begin
       dbms_obfuscation_toolkit.des' || g_RawFunction || 'decrypt
        (input => :1, key => :2, decrypted_data => :3 ' ||
       g_rawWhich ||  ' );
     end;'
     using IN p_data, IN hextoraw(g_charkey), IN OUT l_string;

    return unpadraw(l_string);
end;
```

Функции **DECRYPTSTRING** и **DECRYPTRAW** работают аналогично представленным ранее функциям **ENCRYPT**. Единственное отличие в том, что они вызывают из пакета **DBMS_OBFUSCATION_TOOLKIT** процедуры **DECRYPT** вместо **ENCRYPT** и

используют функцию **UNPAD** для декодирования полученной строки или данных типа **RAW**.

Перейдем к функциям для шифрования больших объектов:

```
function encryptLob(p_data in clob,
                    p_key in varchar2) return clob
as
    l_clob      clob;
    l_offset    number default 1;
    l_len       number default dbms_lob.getlength(p_data);
begin
    setkey(p_key);
    dbms_lob.createtemporary(l_clob, TRUE);
    while (l_offset <= l_len)
    loop
        wa(l_clob, encryptString(
            dbms_lob.substr(p_data, g_chunkSize, l_offset)));
        l_offset := l_offset + g_chunksize;
    end loop;
    return l_clob;
end;

function encryptLob(p_data in blob,
                    p_key  in raw) return blob
as
    l_blob      blob;
    l_offset    number default 1;
    l_len       number default dbms_lob.getlength(p_data);
begin
    setkey(p_key);
    dbms_lob.createtemporary(l_blob,  TRUE);
    while (l_offset <= l_len)
    loop
        wa(l_blob, encryptRaw(
            dbms_lob.substr(p_data, g_chunkSize, l_offset)));
        l_offset := l_offset + g_chunksize;
    end loop;
    return l_blob;
end;
```

Это перегруженные функции для данных типа **BLOB** и **CLOB**. Вначале они создают временный большой объект, в который будут записываться зашифрованные данные. Поскольку при шифровании мы изменяем длину строки/данных типа **RAW**, обеспечивая сохранение исходной длины и дополнение до необходимой, делать это "на месте", используя существующий большой объект, не представляется возможным. Например, при наличии большого объекта размером 64 Кбайт мы "увеличим" первые 32 Кбайт. Теперь необходимо сместить остальные 32 Кбайта большого объекта, чтобы обеспечить пространство для записи увеличенного фрагмента данных. Кроме того, это не позволит вызывать данные функции из SQL-операторов, поскольку локатор большого объекта придется передавать в режиме **IN/OUT**, а при наличии параметров в режиме **IN/OUT**

вызывать функцию в SQL-операторах нельзя. Поэтому мы просто копируем зашифрованные данные в новый большой объект, который затем можно использовать где угодно, в том числе в операторе **INSERT** или **UPDATE**.

Для шифрования и кодирования данных большого объекта используется следующий алгоритм. Начиная с байта 1 (**L_OFFSET**), мы шифруем **G_CHUNKSIZE** байт данных. Они добавляются к ранее созданному временному большому объекту. Добавляем к смещению значение **G_CHUNKSIZE** и продолжаем выполнять тело цикла, пока не обработаем весь большой объект. Возвращаем временный большой объект вызывающему.

Теперь перейдем к дешифрованию данных больших объектов:

```
function decryptLob(p_data in clob,
                    p_key  in varchar2 default NULL) return clob
as
    l_clob      clob;
    l_offset    number default 1;
    l_len       number default dbms_lob.getlength(p_data);
begin
    setkey(p_key);
    dbms_lob.createtemporary(l_clob,  TRUE);
    loop
        exit when l_offset > l_len;
        wa(l_clob, decryptString(
                    dbms_lob.substr(p_data, g_chunksize+8, l_offset)));
        l_offset := l_offset + 8 + g_chunksize;
    end loop;
    return l_clob;
end;

function decryptLob(p_data in blob,
                    p_key in raw default NULL) return blob
as
    l_blob        blob;
    l_offset    number default 1;
    l_len       number default dbms_lob.getlength(p_data);
begin
    setkey(p_key);
    dbms_lob.createtemporary(l_blob,  TRUE);
    loop
        exit when l_offset > l_len;
        wa(l_blob,decryptRaw(
                    dbms_lob.substr(p_data, g_chunksize+8, l_offset)));
        l_offset := l_offset + 8 + g_chunksize;
    end loop;
    return l_blob;
end;
```

В этих функциях мы снова, по тем же причинам, что и ранее, используем временный большой объект для дешифрования. На этот раз, однако, есть еще одна причина для использования временного большого объекта. Если не использовать временный большой объект для записи дешифрованных данных, данные будут дешифроваться непосредственно в базе. Последующие операторы SELECT будут выдавать уже дешифро-

ванные данные, если мы не скопируем их в новый большой объект. В данном случае использовать временный большой объект еще важнее.

Проходим в цикле по фрагментам большого объекта. Начав со смещения 1 (с первого байта) большого объекта, выбираем с помощью **SUBSTR** из него **G_CHUNKSIZE+8** байт. Эти 8 байт добавлены к данным функциями **PADSTR/PADRAW** при кодировании. Итак, обрабатываем большой объект фрагментами размером **G_CHUNKSIZE+8** байт, дешифруем данные и добавляем их к временному большому объекту. Этот объект затем возвращается клиенту.

Теперь рассмотрим последнюю часть пакета **CRYPT_PKG** — интерфейс к подпрограммам, реализующим алгоритм **MD5**:

```
function md5str(p_data in varchar2) return checksum_str
is
    l_checksum_str  checksum_str;
begin
    execute immediate
     'begin :x := dbms_obfuscation_toolkit.md5(input_string => :y); end;'
    using OUT l_checksum_str, IN p_data;
    return l_checksum_str;
end;

function md5raw(p_data in raw) return checksum_raw
is
    l_checksum_raw    checksum_raw;
begin
    execute immediate
     'begin :x := dbms_obfuscation_toolkit.md5(input => :y); end;'
    using OUT l_checksum_raw, IN p_data;
    return l_checksum_raw;
end;

function md51ob(p_data in clob) return checksum_str
is
    l_checksum_str  checksum_str;
begin
    execute immediate
     'begin :x := dbms_obfuscation_toolkit.md5(input_string => :y); end;'
    using OUT l_checksum_str, IN dbms_lob.substr(p_data,g_chunksize,1);
    return l_checksum_str;
end;

function md51ob(p_data in blob) return checksum_raw
is
    l_checksum_raw  checksum_raw;
begin
    execute immediate
     'begin :x := dbms_obfuscation_toolkit.md5(input => :y); end;'
    using OUT l_checksum_raw, IN dbms_lob.substr(p_data,g_chunksize,1);

    return l_checksum_raw;
end;

end;
/
```

Эти функции просто передают данные исходным функциям пакета **DBMS_OBFUSCATION_TOOLKIT.** При этом они, правда, не перегружены, что позволяет их использовать непосредственно в SQL-операторах. Следует помнить, что функции MD5 для больших объектов вычисляют контрольные суммы только по первым **G_CHUNKSIZE** байтам данных. Это связано с ограничением языка PL/SQL на максимальный размер переменных.

Теперь я продемонстрирую функциональные возможности пакета. Следующие примеры выполнялись на сервере Oracle 8.1.7. Такие примеры с использованием алгоритмов DES3 и MD5 в версии 8.1.6 не выполнятся.

```
tkyte@DEV817> declare
  2     l_str_data    varchar2(25) := 'hello world';
  3     l_str_enc     varchar2(50);
  4     l_str_decoded varchar2(25);
  5
  6     l_raw_data    raw(25) :=utl_raw.cast_to_raw('Goodbye');
  7      l_raw_enc     raw(50);
  8      l_raw_decoded raw(25);
  9
 10  begin
 11       crypt_pkg.setkey('MagicKey');
 12
 13      l_str_enc       := crypt_pkg.encryptString(l_str_data);
 14       l_str_decoded  := crypt_pkg.decryptString(l_str_enc);
 15
 16      dbros_output.put_line('Encoded In hex = ' ||
 17                            utl_raw.cast_to_raw(l_str_enc));
 18    dbms_output.put_line('Decoded = ' || l_str_decoded);
 19
 20   crypt_pkg.setkey(utl_raw.cast_to_raw('MagicKey'));
 21
 22      l_raw_enc       := crypt_pkg.encryptRaw(l_raw_data);
 23       l_raw_decoded  := crypt_pkg.decryptRaw(l_raw_enc);
 24
 25      dbms_output.put_line('Encoded= ' || l_raw_enc);
 26    dbms_output.put_line('Decoded=     ' ||
 27                            utl_raw.cast_tq_varchar2(l_raw_decoded));
 28  end;
 29  /
Encoded In hex = 7004DB310AC6A8F210F8467278518CF988DF554B299B35EF
Decoded = hello world
Encoded = E3CC4E04EF3951178DEB9AFAE9C99096
Decoded = Goodbye

PL/SQL procedure successfully completed.
```

Этот пример демонстрирует базовые возможности функций **ENCRYPT** и **DECRYPT**. Здесь я вызывал их из **PL/SQL**, ниже мы будем вызывать их в SQL-операторах. Я протестировал работу со строками и данными типа **RAW**. В строке 11 кода я вызываю процедуру **SETKEY** для установки ключа шифрования, который будет использоваться при шифровании и дешифровании данных типа **VARCHAR2**, равным строке **MAGICKEY**.

Это позволяет не передавать строку остальным функциям. Затем я шифрую строку и помещаю результат в переменную **L_STR_ENC.** После этого строка дешифруется, чтобы убедиться, что все работает, как предполагалось. В строках 16-18 выдаются результаты. Поскольку в зашифрованных данных могут содержаться символы, "сводящие с ума" эмуляторы терминалов, я выдаю зашифрованную строку на терминал, вызвав **UTL_RAW.CAST_TO_RAW** в строке 17. Тип данных меняется с **VARCHAR2** на **RAW.** Сами данные при этом не меняются. Поскольку данные типа **RAW** неявно преобразуются в строку шестнадцатиричных цифр, этот прием можно использовать как удобный способ отображения на экране данных в шестнадцатиричном виде.

В строках с 20 по 27 я делаю то же самое для данных типа **RAW.** Снова необходимо вызвать процедуру **SETKEY,** передав на этот раз 8 байт данных типа **RAW.** Для преобразования ключа типа **VARCHAR2** в ключ типа **RAW** я использовал функцию **UTL_RAW.CAST_TO_RAW.** Можно было также воспользоваться функцией **HEXTORAW** и передать строку шестнадцатиричных цифр. Затем я шифрую данные и дешифрую результат. Зашифрованные данные выдаются в явном виде (они все равно будут отображаться в шестнадцатиричном представлении) и привожу тип расшифрованных данных снова к **VARCHAR2,** чтобы можно было проверить корректность дешифровки. Результат подтверждает, что пакет работает.

Рассмотрим, как этот пакет использовать в языке SQL. Для этого протестируем процесс шифрования тройным DES в режиме с двойным ключом:

```
tkyte@DEV817>drop table t;

Table dropped.

tkyte@DEV817>create table t

  2 (id int primary key, data varchar2(255));

Table created.

tkyte@DEV817>insert into t values
  2 (1, crypt_pkg.encryptString('This is row 1', 'MagicKeyIsLonger'));
1 row created.

tkyte@DEV817>insert into t values
  2  (2, crypt_pkg.encryptString( 'This is row2', 'MagicKeyIsLonger'));

1 row created.

tkyte@DEV817> select utl_raw.cast_to_raw(data) encrypted_in_hex,
  2    crypt_pkg.decryptString(data,'MagicKeyIsLonger')    decrypted
  3    from t
  4  /

ENCRYPTED_IN_HEX                                    DECRYPTED

0B9A809515519FA6A34F15094lB318OA441FBB0C790E9481 This is row 1
0B9A809515519FA6A34F150941B318DA20A936F9848ADC13 This is row 2
```

Итак, задав 16-байтовый ключ процедуре **CRYPT_PKG.ENCRYPTSTRING**, мы автоматически переключились на использование процедуры **DES3ENCRYPT** пакета **DBMS_OBFUSCATIO**N_TOOLKIT. Этот пример показывает, насколько легко использовать средства пакета CRYPT_PKG в языке SQL. Все функции можно непосредствен-

но вызывать в SQL-операторах в тех же конструкциях, что и, например, функцию **SUBSTR.** Средства пакета **CRYPT_PKG** можно использовать в конструкции **SET** оператора **UPDATE,** в конструкции **VALUES** оператора **INSERT,** в списке выбора оператора **SELECT** и даже в конструкции **WHERE** любого оператора.

Теперь посмотрим, как этот пакет можно использовать для больших объектов, продемонстрировав по ходу использование функций MD5. Для проверки используем объект типа **CLOB** размером 50 Кбайт. Сначала загружаем большой объект в базу данных:

```
tkyte@DEV817> create table demo (id int, theClob clob);

Table created.

tkyte@DEV817> create or replace directory my_files as
  2                                 '/d01/home/tkyte';

Directory created.

tkyte@DEV817>declare
  2      l_clob    clob;
  3      l_bfile    bfile;
  4  begin
  5     insert into demo values (1, erapty_clob())
  6      returning theclob into l_clob;
  7
  8      l_bfile :=bfilename('MY_FILES', 'htp.sql');
  9     dbms_lob.fileopen(l_bfile);
 10
 11     dbms_lob.loadfromfile(l_clob, l_bfile,
 12                           dbms_lob.getlength(l_bfile));
 13
 14     dbms_lob.fileclose(l_bfile);
 15  end;
 16  /

PL/SQL procedure successfully completed.
```

Процедуры загрузили данные в объект типа **CLOB**. Теперь мы хотели бы выполнить с ним какие-нибудь действия. Снова используем язык SQL, поскольку это наиболее оптимальный способ работы с данными. Начнем с вычисления контрольной суммы по первым 32 Кбайтам данных объекта типа **CLOB**:

```
tkyte@DEV817>select dbms_lob.getlength(theclob) lob_len,
  2       utl_raw.cast_to_raw(crypt_pkg.md51ob(theclob)) md5_checksum
  3    from demo;

LOB_LENMD5_CHECKSUM

 50601 307D19748889C2DEAD879F89AD45D1BA
```

Для того чтобы преобразовать перед выводом данные типа **VARCHAR2**, возвращаемые функциями MD5, в шестнадцатиричную строку, мы воспользовались функцией **UTL_RAW.CAST_TO_RAW**. Строка типа **VARCHAR2** скорее всего будет содержать встроенные символы новой строки, табуляции или другие управляющие символы терминала. Представленный выше код показывает, насколько легко использовать функции MD5: достаточно передать им данные, и контрольная сумма будет вычислена.

Далее я продемонстрирую, как шифровать и дешифровать большой объект. Для этого используется простой оператор **UPDATE.** Обратите внимание, что на этот раз применяется ключ шифрования длиной 24 байта. Мы будем использовать подпрограмму **DES3ENCRYPT,** поскольку установили необязательный параметр **which => ThreeKeyMode. В** результате будет выполнено шифрование тройным алгоритмом DES с тройным ключом:

```
tkyte@DEV817>update demo
  2            set theClob = crypt_pkg.encryptLob(theClob,
  3                                    'MagicKeyIsLongerEvenMore')
  4   where id = 1;

1 row updated.

tkyte@DEV817>select dbms_lob.getlength(theclob) lob_len,
  2       utl_raw.cast_to_raw(crypt_pkg.md51ob(theclob)) md5_checksum
  3    from demo;

   LOB_LEN MD5_CHECKSUM

     50624 FCBD33DA2336C83685B1A62956CA2D16
```

Длина объекта увеличилась с 50601 до 50624 байт, и рассчитанная по алгоритму MD5 контрольная сумма отличается от прежней — т.е. данные изменены. Если вспомнить представленное ранее описание алгоритма, мы взяли первые 32000 байт объекта типа **CLOB**, добавили в начале 8 байт при кодировании строки и зашифровали результат. Затем мы выбрали оставшиеся 18601 байт данных и дополнили их до 18608 байт (чтобы длина была кратна 8), и добавили еще 8 байт для сохранения исходной длины. Это и дало в результате увеличение длины до 50624 байт.

Теперь рассмотрим, как выбрать из базы данных зашифрованный объект типа CLOB:

```
tkyte@DEV817>select dbms_lob.substr(
  2                       crypt_pkg.decryptLob(theClob), 100, 1) data
  3    from demo
  4   where id = 1;

DATA

set define off
create or replace package htp as
/* STRUCTURE tags */
procedure html0pen;
procedure
```

Интересно отметить, что я не передавал ключ дешифрования. Поскольку он сохранен в переменной пакета, то передавать его необязательно. Значение переменной пакета сохраняется между вызовами, но только на время сеанса. Ключ хранится в глобальной переменной в теле пакета и недоступен другим сеансам.

Проблемы

Данные, зашифрованные средствами пакета **DBMS_OBFUSCATION_TOOLKIT** в системе с обратным порядком байтов ("little endian" system) не могут быть дешифрова-

ны с помощью того же ключа в системе с прямым порядком байтов ("big endian" system). Речь идет о порядке байтов в многобайтовом числе. На Intel-платформах (NT, многие дистрибутивы Linux и Solaris x86) принят обратный порядок байтов. Системы на базе процессоров Sparc и Risc обычно имеют прямой порядок байтов. Данные, зашифрованные на Windows NT с помощью ключа "12345678", нельзя расшифровать на Sparc Solaris с помощью этого же ключа. Следующий пример демонстрирует проблему (и способ ее избежать). На платформе Windows NT выполним:

```
tkyte@TKYTE816>create table anothert (encrypted_data varchar2(25));

Table created.

tkyte@TKYTE816>insert into anothert values
  2    (crypt_pkg.encryptString( 'helloworld', '12345678'));

1 row created.

tkyte@TKYTE816> select crypt_pkg.decryptstring(encrypted_data)
                                                    from anothert;
CRYPT_PKG.DECRYPTSTRING(ENCRYPTED_DATA)

hello world
tkyte@TKYTE816>host exp userid=tom/kyte tables=anothert
```

Соответствующий файл EXPDAT.DMP передан по FTP на машину Sparc Solaris и загружены находящиеся в нем данные. При попытке выбрать данные я получил:

```
ops$tkyte@DEV816> select
  2 crypt_pkg.decryptstring(encrypted_data, '12345678')
  3  from t;
crypt_pkg.decryptstring(encrypted_data, '12345678')
*
ERROR at line 2:
ORA-06502: PL/SQL: numeric or value error: character to number convers
ORA-06512: at "OPS$TKYTE.CRYPT_PKG", line 84
ORA-06512: at "OPS$TKYTE.CRYPT_PKG", line 215
ORA-06512: at line 1

ops$tkyte@DEV816> select
  2 crypt_pkg.decryptstring(encrypted_data, '43218765')
  3  from t;

CRYPT_PKG.DECRYPTSTRING(ENCRYPTED_DATA,'43218765')

hello world
```

Представленное выше сообщение об ошибке выдается пакетом-оболочкой. Я предполагаю, что первые 8 байт данных в строке — это число. Поскольку с помощью переданного ключа нельзя успешно дешифровать данные, первые 8 байт не содержат значение длины — это произвольный набор символов.

Оказывается, 8 байтовый (или 16-, или 24-байтовый ключ) внутренне хранится как набор 4-байтовых целых чисел. Мы должны изменить порядок байтов в каждой 4-байтовой группе символов ключа, чтобы расшифровать данные в системе с другим поряд-

ком байтов. Поэтому, если использовался ключ **'12345678'** на платформе Windows NT (Intel), я должен использовать ключ **'43218765'** на Sparc Solaris. Задаем первые 4 байта в обратном порядке, затем задаем в обратном порядке следующее 4 байта (и так далее — для ключей большего размера).

Это важно помнить при переносе данных, например, с NT на Sparc Solaris и при запросе данных из удаленной базы. Вы должны быть готовы физически переупорядочить байты, чтобы успешно дешифровать данные. Эта проблема была решена в версиях, начиная с Oracle 8.1.7.1, так что теперь переставлять байты уже не нужно.

Управление ключами

Я бы хотел кратко рассмотреть проблемы управления ключами. Шифрование — лишь часть действий, обеспечивающих защиту данных. Данных в базе шифруются для того, чтобы администратор базы данных, выполнив запрос к таблице, не смог понять, какие данные в ней находятся. Например, вы поддерживаете Web-сайт, где принимаете заказы клиентов. Клиенты передают номера кредитных карточек, которые сохраняются в базе данных. Необходимо гарантировать, что ни администратор базы данных, у которого есть возможность выполнять ее резервное копирование, ни злонамеренный хакер, взломавший базу данных, не смогли бы прочитать эту строго конфиденциальную информацию. Если хранить данные в явном виде, их легко сможет прочитать любой, получивший привилегии доступа администратора к базе данных. Если же данные хранятся в зашифрованном виде, этого не произойдет.

Зашифрованные данные защищены настолько, насколько защищен ключ, использовавшийся для шифрования. Тут все дело в ключе. Если ключ известен, данные с таким же успехом можно вовсе не шифровать (при наличии ключа данные можно расшифровать оператором **SELECT).**

Поэтому проблему генерации и защиты ключей следует хорошо продумать. Можно использовать различные подходы. Далее представлено несколько подходов, которые можно использовать, но с каждым из них связаны специфические проблемы.

Генерация и хранение ключей в клиентском приложении

Можно вообще не хранить ключей в базе данных, вынеся их на другую машину (главное, не потерять их — потребуются сотни лет процессорного времени, чтобы их подобрать). При этом клиентское приложение, будь то сервер приложений или клиент в приложении с клиент-серверной архитектурой, сохраняет ключи в своей системе. Клиентское ПО определяет, имеет ли право обратившийся пользователь дешифровать данные, и посылает соответствующие ключи серверу базы данных.

При использовании этого метода, связанного с передачей ключей по сети, необходимо добавить еще один уровень шифрования — шифрование потока данных протокола Net8. Связываемые переменные и строковые литералы по умолчанию передаются в явном виде. В этом случае, поскольку защита ключей принципиально важна, придется использовать технологии типа ASO (Advanced Security Option — расширенная защита).

Эта возможность протокола Net8 обеспечивает шифрование всего потока данных, так что никто не сможет перехватить ключи при передаче по сети.

Если ключ безопасно хранится в клиентском приложении (это должны обеспечить вы сами) и используется **ASO**, это решение будет вполне надежным.

Хранение ключей в той же базе данных

Предполагается хранение ключей вместе с данными в базе. Это решение — не идеально, поскольку есть вероятность, что при наличии достаточного времени администратор базы данных (или хакер, получивший привилегии учетной записи администратора) сможет найти ключи и получить зашифрованные с их помощью данные. В подобных случаях следует максимально затруднить поиск ключей, соответствующих данным. Сделать это сложно, поскольку и ключи, и данные хранятся в одной базе.

Приложение не должно напрямую связывать таблицу ключей с таблицей данных. Предположим, имеется таблица со столбцами **CUSTOMER_ID, CREDIT_CARD** и другими данными. Столбец **CUSTOMER_ID** неизменен — это первичный ключ (а мы знаем, что изменять первичный ключ не стоит). Можно создать другую таблицу:

```
ID   number primary key,
DATA varchar2(255)
```

В этой таблице будут храниться ключи для всех идентификаторов клиентов. Создадим функцию в пакете, которая будет возвращать ключ, только при условии, что ее вызывает **соответствующий** пользователь и в **соответствующей** среде (аналогично тому, как при использовании средств тщательного контроля доступа данные можно получить только в соответствующем контексте приложения).

Пакет будет предоставлять две основные функции.

- **Функция 1: добавление нового клиента. В** данном случае функция будет выполнять некоторые действия с идентификатором клиента, чтобы скрыть его (преобразовать в другую строку). Функция должна быть детерминированной, чтобы по заданному идентификатору клиента всегда выдавалась одна и та же строка. Чуть позже мы поговорим о том, что делать с идентификатором клиента или любой другой строкой. Для клиента также генерируется случайный ключ. Ниже представлен ряд способов генерации этого ключа. Затем с помощью динамического SQL строка вставляется в таблицу ключей. (Не стоит называть таблицу **KEY_TABLE** или еще как-нибудь, чтобы по имени было понятно ее назначение.)

- **Функция 2: получение ключа для клиента.** Функция принимает идентификатор клиента, пропускает его через ту же детерминированную функцию, что и предыдущая, а затем с помощью динамического SQL находит и возвращает ключ для клиента. Все это выполняется, только если текущий пользователь работает в соответствующей среде.

Динамический SQL используется для того, чтобы нельзя было понять, что пакет используется для управления ключами. Пользователь может обратиться к представлению **ALL_DEPENDENCIES** и выяснить, на какие таблицы статически ссылается пакет. При использовании динамического SQL не будет никакой связи между пакетом и таблицей

ключей. Это не позволит скрыть таблицу ключей от очень умного человека, но максимально затруднит ее поиск.

Теперь о том, как скрыть идентификатор клиента или любой набор неизменных данных в строке. (Первичный ключ для этого подходит только при условии, что всегда остается неизменным.) Для этого имеется множество алгоритмов. Если бы я использовал Oracle 8.1.7, то мог бы послать результат конкатенации этих данных с некоторым постоянным значением (его часто называют "затравкой" — "salt") функциям резюмирования по алгоритму MD5 для получения 16-байтовой контрольной суммы. Именно ее я бы использовал в качестве ключа. В Oracle 8.1.6 я мог бы использовать такое же действие, но передавал бы значение функции **DBMS_UTILITY.GET_HASH_VALUE** с очень большим размером хеш-таблицы. Можно было бы применить операцию **XOR** после изменения порядка байтов в **CUSTOMER_ID.** Подойдет любой алгоритм, который сложно угадать по полученному результату.

Вы можете возразить, что администратор базы данных может прочитать код, увидеть алгоритм и разобраться во всем этом. Нет, если **скрыть** (wrap) код. Скрыть PL/SQL-код очень просто (см. описание утилиты **WRAP** в руководстве *PL/SQL User's Guide and Reference).* Она берет исходных код и "шифрует" его. В базу данных загружается "зашифрованная" версия кода. Теперь код никто прочитать не сможет. Средств "дешифрации" для **wrap** нет. Сохраните только в каком-либо безопасном месте текст алгоритма. Восстановить текст после применения утилиты **wrap и** получить исходный код из базы данных не удастся.

Для генерации ключа необходим своего рода генератор случайных строк. Его можно создать разными способами. Можно использовать те же приемы, что и для сокрытия значения **CUSTOMER_ID.** Можно использовать реальный генератор случайных чисел (например, пакет **DBMS_RANDOM** или собственной разработки). Задача в том, чтобы генерировать значение, которое будет сложно "угадать" на основе имеющейся информации.

Лично я предпочел бы хранить ключи именно в базе данных. Если ключи находятся в клиентском приложении, всегда остается риск их потери вследствие сбоя носителя или другой катастрофы в системе. При хранении ключей в файловой системе риск остается. Только хранение ключей в базе данных гарантирует, что зашифрованные данные всегда удастся расшифровать: база данных всегда синхронизирована, а процедуры резервного копирования и восстановления — надежны.

Хранение ключей в файловой системе сервера базы данных

Ключи шифрования данных можно также хранить в файловой системе сервера и обращаться к ним с помощью внешней процедуры на языке C. Я рекомендую использовать внешнюю процедуру на языке C, поскольку цель состоит в том, чтобы спрятать ключи от администратора базы данных, который обычно имеет доступ к учетной записи владельца программного обеспечения Oracle. Пакет **UTL_FILE,** работа с объектами типа **BFILE** и вызов хранимых процедур на языке Java, выполняющих ввод-вывод, осуществляется с правами пользователя-владельца программного обеспечения Oracle. Если

администратор базы данных работает с правами владельца программного обеспечения Oracle, он может прочитать файлы. А прочитав, сможет найти в них ключи. При использовании внешней процедуры, написанной на языке C, можно запустить службу **EXTPROC** (и соответствующий процесс прослушивания для службы **EXTPROC)** от имени другого пользователя. В этом случае пользователь Oracle не увидит ключи. К ним можно получить доступ только через процесс прослушивания **EXTPROC.** Это добавляет еще один уровень защиты. Подробнее о реализации этого подхода см. в главе 18.

Резюме

Мы достаточно подробно рассмотрели пакет **DBMS_OBFUSCATION_TOOLKIT.** Я научил вас создать для него пакет-оболочку, предоставляющий те же функциональные возможности нужным образом (если моя реализация вам не подходит, напишите другую оболочку). Вы научились использовать динамический SQL для создания пакетов, которые можно устанавливать на серверах с разными возможностями (речь шла о возможностях шифрования в версиях 8.1.6 и 8.1.7). Мы обсудили проблему переноса данных на другую платформу при использовании пакета **DMBS_OBFUSCATION_TOOLKIT,** связанную с изменением порядка байтов в ключах. Вы узнали о возможности решать эту проблему переупорядочением байтов ключа. Интересным расширением пакета **CRYPT_PKG** было бы автоматическое определение порядка байтов в системе и перестановка байтов в ключе, чтобы избавить пользователя от необходимости учитывать это различие. Эта идея становится еще более привлекательной, если учесть, что в версиях начиная с 8.1.7.1 менять порядок следования байтов больше не нужно — соответствующий код в этой версии можно не выполнять, что обеспечивает одинаковые функциональные возможности во всех версиях сервера.

Наконец, мы рассмотрели важную проблему управления ключами. Я потратил немало времени на разработку удобного пакета-оболочки, упрощающего шифрование и дешифрование данных. Вам для самостоятельного решения, однако, я оставляю самую сложную проблему — защиту ключей. Следует помнить, что при подозрении взлома ключей необходимо создать новый их набор, расшифровать и снова зашифровать все имеющиеся данные. Если все продумать заранее, подобных ситуаций можно избежать.

Пакет DBMS_OUTPUT

Пакет **DBMS_OUTPUT** пользователи часто понимают неправильно. Они не понимают, что и как он делает и с какими ограничениями связано его использование. В этом разделе я попытаюсь объяснить все это. Я также предложу альтернативные пакеты с возможностями, аналогичными тем, что предоставляются пакетом **DBMS_OUTPUT**, но без упомянутых ограничений.

Пакет **DBMS_OUTPUT** предназначен для эмуляции простых действий вывода на экран в языке PL/SQL. Он позволяет эмулировать выдачу на экран строки Hello World из PL/SQL-кода. Вы уже видели сотни примеров использования этого пакета. Вот типичный пример:

```
ops$tkyte@DEV816> exec dbms_output.put_line('Hello World');
Hello World

PL/SQL procedure successfully completed.
```

Однако вы не видите команды SQL*Plus (или SVRMGRL), которая необходима, чтобы этот пример сработал. Вывод на экран можно включать и отключать следующим образом:

```
ops$tkyte@DEV816> set serveroutput off
ops$tkyte@DEV816> exec dbms_output.put_line('Hello World');

PL/SQL procedure successfully completed.

ops$tkyte@DEV816> set serveroutput on
ops$tkyte@DEV816> exec dbms_output.put_line('Hello World');
Hello World

PL/SQL procedure successfully completed.
```

На самом деле язык PL/SQL не позволяет выполнять ввод-вывод на экран (вот почему я написал, что пакет предназначен для **эмуляции** такой возможности). На самом деле ввод-вывод на экран выполняет утилита SQL*Plus — из языка PL/SQL ничего нельзя выдать на терминал. PL/SQL-код выполняется другим процессом, обычно работающим на другой машине в сети. Утилиты SQL*Plus, **SVRMGRL** и другие инструментальные средства, однако, могут выдавать результаты на экран весьма просто. Это легко обнаружить при использовании пакета **DBMS_OUTPUT** в Java-коде или в программе на Pro*C (или в любой другой программе) — выданные пакетом **DBMS_OUTPUT** результаты никогда не выдаются на экран. Дело в том, что приложение само отвечает за выдачу этих результатов.

Как работает пакет DBMS_OUTPUT

В пакете DBMS_OUTPUT подпрограмм немного. Чаще всего используются следующие.

- **PUT**. Выдает строку, данные типа **NUMBER** или **DATE** в буфер вывода, не добавляя символ новой строки.
- **PUT_LINE**. Выдает строку, данные типа **NUMBER** или **DATE** в буфер вывода и добавляет символ новой строки.
- **NEW_LINE**. Выдает в буфер вывода символ новой строки.
- **ENABLE/DISABLE**. Включает и отключает выдачу в буфер, используя **DBMS_OUTPUT**.

Эти процедуры выдают данные во внутренний буфер; PL/SQL-таблицу в теле пакета **DBMS_OUTPUT**. Общая длина выдаваемой строки (сумма всех байтов, помещенных в буфер пользователем до вызова процедуры **PUT_LINE** или **NEW_LINE**, завершающей эту строку) должна быть не более 255 байт. Выданные результаты буферизуются в этой таблице и не будут видны в среде SQL*Plus, пока не завершится выполнение соответствующего PL/SQL-кода. Язык PL/SQL не позволяет ничего выдавать на терминал, данные просто помещаются в PL/SQL-таблицу.

При вызове процедуры **DBMS_OUTPUT.PUT_LINE** пакет **DBMS_OUTPUT** сохраняет соответствующие данные в массиве (PL/SQL-таблице) и возвращает управление. Пока работа вызывающей процедуры не завершится, результаты на экран не выдаются. Даже после этого результаты можно увидеть, только если используемый клиент "знает" о пакете **DBMS_OUTPUT** и сам позаботится о выдаче накопленных результатов. Утилита SQL*Plus, например, вызывает **DBMS_OUTPUT.GET_LINES** для получения части содержимого буфера пакета **DBMS_OUTPUT** и выдачи его на экран. Если вызвать хранимую процедуру из приложения на языке Java/JDBC, предположение о том, что результаты **DBMS_OUTPUT** окажутся там же, где и данные, выданные с помощью функции System.out.println, не оправдается. Если клиентское приложение не позаботится о выборке и выдаче данных, они просто исчезнут. Как сделать это в программе на языке Java/JDBC будет продемонстрировано в этом разделе.

При использовании пакета **DBMS_OUTPUT** больше всего сбивает с толку то, что результат буферизуется и не выдается до завершения процедуры. Пользователи зна-

ют о пакете **DBMS_OUTPUT** и пытаются использовать его для контроля продолжительного процесса. Другими словами, они повсеместно вставляют в код вызовы **DBMS_OUTPUT.PUT_LINE** и запускают процедуру в среде SQL*Plus. Они ждут, что результаты начнут выдаваться на экран, и очень расстраиваются, когда этого не происходит (потому что и не может произойти). Не зная особенностей реализации, трудно понять, почему результаты не выдаются сразу. Если понять, что процедуры на языке PL/SQL (а также внешние процедуры на языках Java и C), работающие на сервере, не выполняют ввод-вывод на экран и что пакет **DBMS_OUTPUT** просто накапливает данные в большом массиве, ситуация проясняется. Вот когда имеет смысл вернуться к разделу, посвященному пакету **DBMS_APPLICATION_INFO, и** почитать о способе контроля работы продолжительных действий. Для контроля продолжительно работающих процессов надо использовать пакет **DBMS_APPLICATION_INFO,** а не **DBMS_OUTPUT.**

Для чего же тогда пакет **DBMS_OUTPUT?** Он прекрасно подходит для выдачи простых отчетов и создания утилит. В главе 23 была представлена процедура **PRINT_TABLE,** использующая средства пакета **DBMS_OUTPUT** для генерации результатов следующего вида:

```
SQL> exec print_table('select * from all_users where username = user');
USERNAME                       : OPS$TKYTE
USER_ID                        : 334
CREATED                        : 02-oct-2000 10:02:12

PL/SQL procedure successfully completed.
```

Она выдает данные **по одному столбцу в строке,** а не одной, разбитой на части, строкой. Прекрасно подходит для выдачи длинных строк таблиц, которые переносятся по границе экрана, что затрудняет чтение.

Теперь, зная, что пакет **DBMS_OUTPUT** работает путем помещения данных в PL/SQL-таблицу, можно изучать его реализацию. При включении **DBMS_OUTPUT** (вызовом **DBMS_OUTPUT.ENABLE** либо с помощью команды **SET SERVEROUTPUT ON**) мы не только позволяем накапливать данные, но и задаем максимальный объем данных, которые сможем накопить. По умолчанию, если выполнить:

```
SQL> set serveroutput on
```

создается буфер **DBMS_OUTPUT** размером 20000 байт. При достижении этого предела будет выдано:

```
begin
*
ERROR at line 1:
ORA-20000: ORU-10027: buffer overflow, limit of 20000 bytes
ORA-06512: at "SYS.DBMS_OUTPUT", line 106
ORA-06512: at "SYS.DBMS_OUTPUT", line 65
ORA-06512: at line 3
```

Этот предел можно увеличить, выполнив команду **SET SERVEROUTPUT** (или вызвав процедуру **DBMS_OUTPUT.ENABLE**):

```
SQL> set serveroutput on size 1000000

SQL> set serveroutput on size 1000001
SP2-0547: size option 1000001 out of range (2000 through 1000000)
```

Как видно из полученного сообщения об ошибке, однако, размер буфера должен быть в диапазоне от 20000 до 1000000 байт. Реально вы сможете поместить в буфер **меньше** данных, чем установленный предел. Пакет **DBMS_OUTPUT** использует простой алгоритм упаковки данных при помещении в PL/SQL-таблицу. Он не помещает i-ю строку в i-ый элемент массива, он плотно упаковывает массив. В первом элементе массива может оказаться первых пять строк, выданных с помощью этого пакета. Для этого (чтобы поместить несколько строк в одну) необходимо расходовать дополнительные ресурсы. Эти дополнительные ресурсы для размещения сведений о том, где находятся данные пользователя и какого они размера, выделяются из того же ограниченного пространства. Поэтому, даже выполнив команду **SET SERVEROUTPUT ON SIZE 1000000,** вы сможете выдать меньше миллиона байтов.

Можно ли определить, сколько байтов можно будет выдать? Иногда — да, а иногда — нет. При фиксированном размере выдаваемой строки, когда все строки одинаковой длины, это можно определить. Можно точно рассчитать, сколько байтов удастся выдать. Если же выдаются строки переменной длины, то вычислить заранее, сколько байтов удастся выдать, нельзя. Ниже я представлю алгоритм, используемый сервером Oracle для упаковки данных.

Мы знаем, что сервер Oracle сохраняет данные в массиве. Максимальное количество строк в этом массиве рассчитывается исходя из установки SET **SERVEROUTPUT** ON SIZE. Массив пакета **DBMS_OUTPUT** никогда не будет длиннее **IDXLIMIT** строк, где **IDXLIMIT** рассчитывается как:

```
idxlimit := trunc((xxxxxx+499) / 500);
```

Итак, если выполнить **SET SERVEROUTPUT ON SIZE 1000000**, пакет DBMS_OUTPUT будет использовать не более 2000 элементов массива. Пакет DBMS_OUTPUT будет сохранять в каждом элементе массива не более 504 байт данных (обычно — меньше). Пакет DBMS_OUTPUT упаковывает данные в строку массива в следующем формате:

```
Буфер(1) = '<sp>NNNваши данные<sp>NNNваши данные...';
Буфер(2) = '<sp>NNNваши данные<sp>NNNваши данные...';
```

Так что для каждой выдаваемой строки будет использоваться дополнительно 4 байта — для одного пробела и трехзначного числа. Каждая строка в буфере **DBMS_OUTPUT** имеет длину не более **504** байт, и пакет **DBMS_OUTPUT** не будет переносить данные с одной строки на другую. Поэтому, например, если использовать максимальную длину строки и всегда выдавать строки по 255 байт, пакет **DBMS_OUTPUT** сможет упаковать в элемент массива только одну строку. Причина в том, что значение **(255+4) * 2 = 518** больше, чем **504,** а пакет **DBMS_OUTPUT** не будет делить строку на два элемента своего массива. Две строки такого размера просто не помещаются в одну строку массива **DBMS_OUTPUT.** Поэтому даже если затребовать буфер размером 1000000 байт, вы сможете поместить в него только 510000 байт данных — чуть больше **половины** запрошенного. Значение 510000 получено, исходя из того, что длина выдаваемых строк — 255 байт;

всего же строк может быть не более 2000 (вспомните представленное ранее вычисление значения **IDXLIMIT); 255*2000 = 510000.** С другой стороны, при использовании строк длиной 248 байт можно помещать по две строки в элемент массива, что позволит выдать **248 * 2 * 2000 = 992000** байт — чуть больше 99% запрошенного пространства. Фактически, это максимум того, на что можно рассчитывать при использовании пакета **DBMS_OUTPUT** — 992000 байт данных. Больше выдать с помощью этого пакета нельзя.

Как уже было сказано, при использовании строк фиксированной длины очень легко подсчитать выдаваемое количество строк. Если известна фиксированная длина строки, например 79, **80** или 81 байт, легко все рассчитать:

```
ops$tkyte@ORA8I.WORLD> select trunc(504/(79+4)) * 79 * 2000 from dual;

TRUNC(504/(79+4))*79*2000

                948000

ops$tkyte@ORA8I.WORLD> select trunc(504/(80+4)) * 80 * 2000 from dual;

TRUNC(504/(80+4))*80*2000

                960000

ops$tkyte@ORA8I.WORLD> select trunc(504/(81+4)) * 81 * 2000 from dual;

TRUNC(504/(81+4))*81*2000

                810000
```

Как видите, максимальный объем выдаваемых данных сильно зависит от размера выдаваемой строки.

Проблема со строками переменной длины состоит в том, что максимальный объем результата предсказать нельзя. Он зависит от того, как идет выдача, от последовательности строк, получаемых пакетом **DBMS_OUTPUT.** Если выдавать одни и те же строки, но в другом порядке, их будет выдано больше или меньше. Это непосредственно связано с используемым алгоритмом упаковки.

Эта особенность пакета **DBMS_OUTPUT** сбивает с толку. Вы можете выполнить процедуру один раз и успешно выдать отчет размером 700000 байт, а завтра та же процедура приведет к выдаче сообщения об ошибке **ORA-20000: ORU-10027: buffer overflow** после получения 650000 байт. Это связано со способом упаковки данных в буфере пакета **DBMS_OUTPUT.** Далее в этом разделе мы рассмотрим альтернативы пакету **DBMS_OUTPUT,** позволяющие избежать этой неоднозначности.

Резонно задать вопрос: а зачем создатели пакета вообще делают эту упаковку? Причина в том, что, когда пакет **DBMS_OUTPUT** появился в версии 7.0, выделение памяти для PL/SQL-таблиц выполнялось совсем не так, как сейчас. При выделении слота в PL/SQL-таблице сразу же выделялась память для элемента максимального размера. Это означает, что, поскольку **DBMS_OUTPUT** использует элементы типа **VARCHAR2(500),** 500 байт будут выделены при вызове **DBMS_OUTPUT.PUT_LINE('hello world'),** т.е. тот же объем, что и при выдаче длинной строки. Результат, состоящий из 2000 строк, занял бы 1000000 байт, даже если выдать 2000 раз строку **hello world,** что фактически требует только около 22 Кбайт. Так что подобная упаковка была предусмотрена для предотвращения выделения лишней памяти в области PGA для буферного массива. В последних

версиях Oracle (начиная с 8.0) память выделяется по-другому. Размер элементов массива меняется динамически, и упаковка больше не нужна. Поэтому ее можно считать унаследованной от старых версий.

Последнее, что хотелось бы сказать о работе пакета **DBMS_OUTPUT,** — это то, что начальные пробелы в выдаваемых строках удаляются. Ошибочно думать, что это "свойство" пакета DBMS_OUTPUT. Фактически, это "свойство" SQL*Plus (хотя, я знаю многих, кто склонен считать это ошибкой). Небольшой тест позволит понять, что я имею в виду:

```
ops$tkyte@ORA8I.WORLD> exec  dbms_output.put_line('  hello world'),
hello world

PL/SQL procedure successfully completed.
```

При передаче пакету **DBMS_OUTPUT** строки ' **hello world',** начальные пробелы оказались удалены. Считается, что это делает пакет **DBMS_OUTPUT,** но на самом деле это не так. Усекает начальные пробелы утилита SQL*Plus. Простое решение этой проблемы — использовать расширенный синтаксис команды SET **SERVEROUTPUT.** Вот полный синтаксис этой команды:

```
set serveroutput {ON|OFF} [SIZE n]
        [FORMAT {WRAPPED|TORD_WRAPPED|TRUNCATED}]
```

Значение конструкций формата выдачи строк представлено ниже.

- **WRAPPED.** Утилита SQL*Plus при необходимости переносит на новую строку выданные сервером результаты, начиная с позиции, задаваемой командой **SET LINESIZE.**
- **WORD_WRAPPED.** Переносится каждая строка выданных сервером результатов, начиная с позиции, задаваемой командой **SET LINESIZE.** Перенос выполняется по словам. Утилита SQL*Plus выравнивает каждую строку влево, **удаляя все начальные пробелы.** Это значение является стандартным.
- **TRUNCATED**. Каждая строка результатов сервера усекается до длины, задаваемой командой **SET LINESIZE.**

Действие каждой опции форматирования проще всего понять на примере:

```
SQL>set linesize 20
SQL>set serveroutput on format wrapped
SQL>exec dbms_output.put_line('        Hello     World        !!!!!');
     Hello     World
      !!!!!

PL/SQL procedure successfully completed.

SQL>set serveroutput on format word_wrapped
SQL>exec     dbms_output.put_line('     Hello     World        !!!!!');
Hello     World
!!!!!

PL/SQL procedure successfully completed.

SQL>set serveroutput on format truncated
SQL>execdbms_output.put_line('        Hello     World        !!!!!');
```

```
     Hello     World

PL/SQL procedure successfully completed.
```

Пакет DBMS_OUTPUT в других средах

Стандартные средства, такие как SQL*Plus и **SVRMGRL,** учитывают особенности работы пакета **DBMS_OUTPUT.** Большинство остальных сред — нет. Например, обычная программа на языке Java/JDBC о пакете **DBMS_OUTPUT** ничего "не знает". В этом подразделе мы рассмотрим, как учесть в такой программе особенности работы пакета **DBMS_OUTPUT.** Такой же подход можно применить в любой среде программирования. Методы, использованные мной для языка Java, можно применить в среде Pro*C, OCI, VB и в любой другой среде.

Начнем с небольшой PL/SQL-процедуры, генерирующей данные для вывода:

```
scott@TKYTE816> create or replace
  2  procedure emp_report
  3  as
  4  begin
  5      dbms_output.put_line
  6    (rpad('Empno',    7) ||
  7       rpad('Ename',12)  ||
  8        rpad('Job',11));
  9
 10      dbms_output.put_line
 11    (rpad('-',    5,  '-') ||
 12    rpad('-',12,'-')    ||
 13     rpad('     -',11,'-'));
 14
 15      for x in (select * from emp)
 16      loop
 17          dbms_output.put_line
 18         (to_char(x.empno, '9999') || '   ' ||
 19           rpad(x.ename, 12)  ||
 20           rpad(x.job, 11));
 21      end loop;
 22  end;
 23  /

Procedure created.

scott@TKYTE816> set serveroutput on format wrapped
scott@TKYTE816> exec emp_report
Empno  Ename       Job

 7369  SMITH       CLERK
 7499  ALLEN       SALESMAN

 7934  MILLER      CLERK

PL/SQL procedure successfully completed.
```

Теперь создадим класс, позволяющий работать с буфером пакета **DBMS_OUTPUT** в среде Java/JDBC:

```
import java.sql.*;

class DbmsOutput
{
/*
 * Переменные экземпляра. Всегда лучше использовать вызываемые,
 * или подготовленные, операторы и готовить (анализировать)
 * их один раз при выполнении программы, а не при каждом выполнении
 * оператора. Повторный анализ требует очень больших ресурсов.
 * Не забудьте также использовать СВЯЗЫВАЕМЫЕ ПЕРЕМЕННЫЕ!
 *
 * В этом классе мы используем три оператора. Один — для включения
 * DBMS_OUTPUT, аналог команды SET SERVEROUTPUT ON в SQL*Plus,
 * второй — для выключения, подобно SET SERVEROUTPUT OFF.
 * Третий — для "сброса" или выдачи результатов вызовов DBMS_OUTPUT
 * с помощью system.out.
 *
 */
private CallableStatement enable_stmt;
private CallableStatement disable_stmt;
private CallableStatement show_stmt;

/*
 * Конструктор готовит три
 * оператора, которые предполагается выполнить.
 *
 * Оператор, который мы готовим для SHOW, — это блок кода
 * для возврата строки результатов DBMS_OUTPUT. Обычно
 * можно использовать тип PL/SQL-таблицы, но JDBC-драйверы
 * не поддерживают типы PL/SQL-таблиц. Поэтому мы получаем результат
 * и конкатенируем его в строку. Будем выбирать не более одной строки
 * результата, так что можем превзойти значение параметра MAXBYTES.
 * Если установить MAXBYTES равным 10, а первая строка имеет длину 100
 * байт, вы получите 100 байт. Параметр MAXBYTES не даст
 * получить следующую строку, но разбиения строки не произойдет.
 *
 */
public DbmsOutput(Connection conn) throws SQLException
{
    enable_stmt  = conn.prepareCall("begin dbms_output.enable(:1); end;");
    disable_stmt = conn.prepareCall("begin dbms_output.disable; end;");
    show_stmt = conn.prepareCall(
          "declare " +
               l_line varchar2(255); " +
          "    l_done number;  " +
          "    l_buffer long;  " +
          "begin " +
          "  loop " +
          "    exit when length(l_buffer)+255 > :maxbytes OR l_done = 1;  " +
```

```
            "   dbms_output.get_line(l_line,  l_done); " +
                  l_buffer := l_buffer || l_line || chr(10); " +
            " end loop;  " +
            " :done := l_done; " +
            " :buffer := l_buffer; " +
            "end;");
>

/*
 * ENABLE задает размер и выполняет
 * вызов DBMS_OOTPOT.ENABLE
 *
 */
public void enable(int size) throws SQLException
{
    enable_stmt.setInt(1,  size);
    enable_stmt.executeUpdate();
)

/*
 * DISABLE просто вызывает DBMS_OUTPUT.DISABLE
 */
public void disable() throws SQLException
{
    disable_stmt.executeUpdate();
)

/*
 * Функция SHOW выполняет основную работу. Она циклически
 * выбирает данные DBMS_OUTPUT,
 * по 32000 байт за раз (плюс-минус 255 байт).
 * По умолчанию она выдает результат в стандартный
 * выходной поток (для перенаправления результатов
 * достаточно перенаправить System.out).
 */
public void show() throws SQLException
{
int               done = 0;

    show_stmt.registerOutParameter(2, java.sql.Types.INTEGER);
    show_stmt.registerOutParameter(3, java.sql.Types.VARCHAR);

    for(;;)
    {
        show_stmt.setInt(1,  32000);
        show_stmt.executeUpdate();
        System.out.print(show_stmt.getString(3));
        if  ((done = show_stmt.getInt(2))  = 1)  break;
    )
}

/*
 * Функция CLOSE закрывает все выполняемые операторы, связанные
 * с классом DbmsOutput. Ее надо вызывать, если вы создали оператор
```

```
 * DbmsOutput и он выходит из области действия, как и для любого
 * вызываемого оператора, результирующего множества и т.п.
 */
public void close() throws SQLException
{
    enable_stmt.close();
    disable_stmt.close();
    show_stmt.close();
}
}
```

Чтобы продемонстрировать использование этого класса, я создал небольшую тестовую программу на языке Java/JDBC. Здесь dbserver — имя сервера базы данных, а ora8i — имя службы, соответствующей экземпляру:

```
import java.sql.*;

class test {

public static void main (String args [])
   throws SQLException
{
    DriverManager.registerDriver
      (new oracle.jdbc.driver.OracleDriver());

    Connection conn = DriverManager.getConnection
         ("jdbc:oracle:thin:6dbserver:1521:ora8i",
          "scott", "tiger");
    conn.setAutoCommit (false);

    Statement stmt = conn.createStatement();

    DbmsOutput dbmsOutput = new DbmsOutput(conn);

    dbmsOutput.enable(1000000);
    stmt.execute
     ("begin emp_report; end;");
    stmt.close();

    dbmsOutput.show();

    dbmsOutput.close();
    conn.close();
}
}
```

Теперь протестируем программу, скомпилировав и выполнив ее:

```
$ javac test.java

$ java test
Empno  Ename      Job

 7369  SMITH      CLERK
 7499  ALLEN      SALESMAN
 7521  WARD       SALESMAN
```

Итак, это показывает, как в языке Java использовать средства пакета **DBMS_OUTPUT.** Как и в случае утилиты SQL*Plus, необходимо вызывать **DbmsOutput.show()** после выполнения оператора, выдающего какие-либо результаты. После выполнения оператора **INSERT, UPDATE, DELETE** или вызова хранимой процедуры утилита SQL*Plus вызывает подпрограмму **DBMS_OUTPUT.GET_LINES** для получения результата. Приложение на языке Java (или C, или VB) должно вызывать функцию **SHOW** для выдачи результатов.

Обход ограничений

Я обнаружил два основных ограничения пакета **DBMS_OUTPUT:**

- Длина "строки" ограничена 255 байтами. Символ новой строки надо вставлять не реже, чем через 255 байт.
- Общий объем выдаваемых результатов ограничен и находится в пределах от 200000 байт (если выдавать по одному байту в строке) до 992000 байт (если выдавать по 248 байт в строке). Для некоторых действий этого хватает, для других - недостаточно, особенно если учесть, что общий объем результатов, которые можно выдать, зависит от длины и порядка выдачи строк.

Итак, что можно сделать? В следующих подразделах я предложу три способа обойти эти ограничения.

Использование небольшой функции-оболочки или другого пакета

Иногда 255 байт не хватает. Необходимо выдать отладочную информацию, и получается строка длиной 500 символов. Ее надо выдать, при этом не так важен формат, как возможность получить результаты. В этом случае можно написать небольшую подпрограмму-оболочку. Во всех моих базах данных установлена такая подпрограмма, позволяющая обойти ограничение длины строки и заодно сократить вызов, поскольку строка **DBMS_OUTPUT.PUT_LINE** состоит из 20 символов, что многовато для постоянного набора. Я часто использую процедуру P. Вот эта процедура:

```
procedure p(p_string in varchar2)
is
    l_string long default p_string;
begin
    loop
      exit when l_string is null;
      dbms_output.put_line(substr(l_string,  1,  248));
      l_string := substr(l_string, 251);
    end loop;
end;
```

Она не переносит переданную строку по словам и вообще ничего особенного не делает. Она принимает строку размером до 32 Кбайт и выдает ее. Длинную строку она разобьет на ряд строк размером 248 байт каждая (248 — оптимальное значение, кото-

рое мы вычислили ранее; оно позволяет выдать максимальный объем результатов). Процедура меняет данные (поэтому она не подходит для увеличения длины строки в процедуре, создающей текстовый файл), выдавая переданную строку в виде нескольких строк.

Процедура решает простую проблему. Она позволяет избавиться от сообщения об ошибке:

```
ops$tkyte@ORA8l.WORLD> exec dbms_output.put_line(rpad('*',256,'*'))
BEGIN dbms_output.put_line(rpad('*', 256,'*')); END;

*
ERROR at line 1:
ORA-20000: ORU-10028: line length overflow, limit of 255 bytes per line
ORA-06512: at "SYS.DBMS_OUTPUT", line 99
ORA-06512: at "SYS.DBMS_OUTPUT", line 65
ORA-06512: at line 1
```

получаемого при выдаче отладочной информации или печати отчета.

Более надежный способ обойти это ограничение, особенно при сбросе данных в текстовый файл, — использовать вместо пакета **DBMS_OUTPUT** средства пакета **UTL_FILE** для записи **непосредственно** в файл. Пакет UTL_FILE ограничивает размер выдаваемой строки 32 Кбайтами и не ограничивает размер файла. С помощью пакета **UTL_FILE** можно создавать файлы только на сервере, так что это решение не подойдет, если предполагается использование утилиты SQL*Plus на клиенте, подключенном по сети, с выдачей результатов в локальный файл на клиенте. Если же необходимо создать текстовый файл данных для загрузки и его создание на сервере допустимо, пакет **UTL_FILE** вполне можно использовать.

Итак, мы рассмотрели две из трех возможностей. Переходим к последней.

Создание аналога пакета DBMS_OUTPUT

Это универсальное решение, хорошо работающее во всех средах. Мы собираемся заново изобрести велосипед, но велосипед более совершенный. Создадим пакет-аналог DBMS_OUTPUT, который:

- ограничивает длину строки 4000 байтами (это, к сожалению, ограничение языка SQL, а не PL/SQL);
- не ограничивает количество выдаваемых строк;
- позволяет получать результаты на клиенте, как и пакет **DBMS_OUTPUT;**
- не позволяет утилите SQL*Plus удалять начальные пробелы в строке, независимо от режима;
- позволяет выбирать результаты как результирующее множество на клиенте с помощью курсора (к результату можно будет делать запросы).

Начнем с создания SQL-типа. Этот тип будет использоваться для буфера, аналогичного тому, что применяется в пакете **DBMS_OUTPUT.** Поскольку это SQL-тип, можно применять к данным операторы **SELECT** *. Поскольку практически в любой среде можно выполнить оператор **SELECT** *, выдать результаты не представит сложности.

```
ops$tkyte@ORA8I.WORLD> create or replace type my_dbms_output_type
  2  as table of varchar2(4000)
  3  /

Type created.
```

Теперь переходим к спецификации пакета-аналога **DBMS_OUTPUT**. Этот пакет устроен подобно **DBMS_OUTPUT**. В нем нет только подпрограмм **GET_LINE** и **GET_LINES**, поскольку в нашей реализации они не нужны. Процедуры **PUT**, **PUT_UNE** и **NEW_LINE** работают точно так же, как их аналоги в пакете DBMS_OUTPUT. Функции **GET**, **FLUSH** и **GET_AND_FLUSH** — новые. Аналогов для них в пакете **DBMS_OUTPUT** нет. Эти функции используются для получения результата после выполнения хранимой процедуры. Функция **GET** будет просто возвращать данные из буфера, но не "стирать" их. Можно вызывать функцию **GET** повторно для получения одного и того же содержимого буфера (пакет **DBMS_OUTPUT** всегда сбрасывает буфер). Функция **FLUSH** позволяет сбросить буфер, другими словами, очистить его. Функция **GET_AND_FLUSH**, как можно догадаться, возвращает содержимое буфера и очищает его; следующие вызванные подпрограммы пакета будут работать с пустым буфером:

```
tkyte@TKYTE816> create or replace package my_dbms_output
  2  as
  3     procedure enable;
  4     procedure disable;
  5
  6      procedure put(s in varchar2);
  7      procedure put_line(s in varchar2);
  8     procedure new_line;
  9
 10     function get return my_dbms_output_type;
 11     procedure flush;
 12    function get_and_flush return my_dbms_output_type;
 13  end;
 14  /

Package created.
```

Используем некоторые из методов, которые рассмотрены в главе 20, посвященной использованию объектно-реляционных средств. В частности, используем возможность выполнять операторы **SELECT * from PLSQL_FUNCTION** — именно так и будет работать аналог пакета **DBMS_OUTPUT**. Наибольший интерес представляют подпрограммы **ENABLE**, **DISABLE**, **PUT**, **PUT_LINE** и **NEW_LINE**. Они работают более-менее похоже на одноименные подпрограммы пакета **DBMS_OUTPUT**. Основное различие в том, что процедура **ENABLE** не имеет параметров, а пакет **MY_DBMS_OUTPUT** по умолчанию выдает результаты (тогда как пакет **DBMS_OUTPUT** по умолчанию их не выдает). Выдаваемые результаты ограничены объемом оперативной памяти, который вы можете выделить в системе (учтите это!). Рассмотрим тело пакета. Реализация этого пакета очень проста. Имеется глобальная переменная пакета, используемая в качестве буфера для результатов. Мы добавляем строки текста в буфер.выделяя при необходимости дополнительную память. Чтобы сбросить буфер, присваиваем ему пустую таблицу. Поскольку все так просто, я представлю реализацию без комментариев:

```
tkyte@TKYTE816> create or replace package body my_dbms_output
  2  as
  3
  4  g_data       my_dbms_output_type  := my_dbms_output_type();
  5  g_enabled    boolean default TRUE;
  6
  7      procedure enable
  8      is
  9      begin
 10          g_enabled := TRUE;
 11      end;
 12
 13      procedure disable
 14      is
 15      begin
 16          g_enabled := FALSE;
 17      end;
 18
 19      procedure put(s in varchar2)
 20      is
 21      begin
 22         if (NOT g_enabled) then return; end if;
 23         if (g_data.count <> 0) then
 24            g_data(g_data.last) :=g_data(g_data.last) || s;
 25         else
 26             g_data.extend;
 27            g_data(1) := s;
 28         end if;
 29      end;
 30
 31      procedure put_line(s in varchar2)
 32      is
 33      begin
 34         if (NOT g_enabled) then return; end if;
 35         put(s);
 36         g_data.extend;
 37      end;
 38
 39      procedure new_line
 40      is
 41      begin
 42         if (NOT g_enabled) then return; end if;
 43         put(null);
 44         g_data.extend;
 45      end;
 46
 47
 48      procedure flush
 49      is
 50         l_empty      my_dbms_output_type  := my_dbms_output_type();
 51      begin
 52         g_data := l_empty;
```

```
 53      end;
 54
 55      function get return my_dbms_output_type
 56      is
 57      begin
 58          return g_data;
 59      end;
 60
 61      function get_and_flush return my_dbms_output_type
 62      is
 63         l_data       my_dbms_output_type := g_data;
 64         l_empty      my_dbms_output_type := my_dbms_output_type();
 65      begin
 66          g_data := l_empty;
 67          return l_data;
 68      end;
 69  end;
 70  /

Package body created.
```

Теперь, чтобы сделать пакет действительно полезным, необходим простой метод получения содержимого буфера. Можно вызывать функции **MY_DBMS_OUTPUT.GET** или **GET_AND_FLUSH** и выбирать содержимое переменной объектного типа самостоятельно или использовать одно из созданных ниже представлений. Первое представление, **MY_DBMS_OUTPUT_PEEK,** обеспечивает SQL-интерфейс к функции **GET.** Оно позволяет многократно запрашивать данные из буфера результатов, фактически обеспечивая просмотр буфера без сброса результатов. Второе представление, **MY_DBMS_OUTPUT_VIEW,** позволяет выполнить запрос к буферу один раз - любые последующие вызовы подпрограмм **PUT, PUT_LINE, NEW_LINE, GET** или **GET_AND_FLUSH** будут работать с пустым буфером результатов. Оператор **SELECT * FROM MY_DBMS_OUTPUT_VIEW** аналогичен вызову функции **DBMS_OUTPUT.GET_LINES.** Буфер сбрасывается:

```
tkyte@TKYTE816> create or replace
  2  view my_dbms_output_peek (text)
  3  as
  4  select *
  5    from TABLE (cast(my_dbms_output.get()
  6                          as my_dbms_output_type))
  7  /

View created.

tkyte@TKYTE816> create or replace
  2  view my_dbms_output_view (text)
  3  as
  4  select *
  5    from TABLE (cast(my_dbms_output.get_and_flush()
  6                          as my_dbms_output_type))
  7  /

View created.
```

Теперь все готово для демонстрации работы этого решения. Выполним процедуру, генерирующую данные в буфер, а затем посмотрим, как их выдать и что с ними можно делать:

```
tkyte@TKYTE816> begin
  2          my_dbms_output.put_line('hello');
  3        my_dbms_output.put('Hey ' ) ;
  4        my_dbms_output.put('there ' ) ;
  5          my_dbms_output.new_line;
  6
  7      for i in 1 .. 20
  8       loop
  9     my_dbms_output.put_line(rpad('    ', i, ' ') || i);
 10       end loop;
 11  end;
 12  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> select *
  2    from my_dbms_output_peek
  3  /

TEXT

hello
Hey there
 1
  2

                  19
                   20

23 rows selected.
```

Интересно, что утилита SQL*Plus, создатели которой ничего не знали о пакете **MY_DBMS_OUTPUT,** не выдает результаты автоматически. Надо ей помочь, выполнив запрос, выдающий результаты.

Поскольку для получения результатов используются SQL-операторы, вы легко сможете написать собственный класс **DbmsOutput** на языке Java/JDBC. Это будет простой объект **ResultSet** — ничего больше. В качестве последнего комментария к этому коду скажу, что результаты ожидают выборки в буфере:

```
tkyte@TKYTE816> select *
  2    from my_dbms_output_peek
  3  /

TEXT

hello
Hey there
 1
  2
```

```
                    19
                     20

23 rows selected.
```

Более того, при выборке можно задавать конструкцию **WHERE**, сортировать результаты, соединять их с другими таблицами и т.д. (как и для данных любой таблицы):

```
tkyte@TKYTE816>select *
  2    from my_dbms_output_peek
  3   where text like '%1%'
  4   /

TEXT

 1
          10
           11

                  18
                   19


11 rows selected.
```

Если же повторное обращение к данным нежелательно, можно выбрать результаты с помощью оператора **SELECT** из представления **MY_DBMS_OUTPUT_VIEW:**

```
tkyte@TKYTE816>select *
  2    from my_dbms_output_view
  3   /

TEXT

hello
Hey there
 1

                   19
                    20


23 rows selected.

tkyte@TKYTE816>select *
  2    from my_dbms_output_view
  3   /

no rows selected
```

В этом случае данные можно получить только один раз.

Эта новая реализация пакета **DBMS_OUTPUT** увеличивает допустимую длину строки с 255 байт до 4000 и фактически снимает ограничение на общий объем выдаваемых результатов (вы, однако, ограничены объемом оперативной памяти сервера). Она также предоставляет ряд новых возможностей (можно делать запросы к результатам, сортиро-

вать их и т.д). Она позволяет избавиться от стандартного удаления начальных пробелов в среде SQL*Plus. Наконец, в отличие от пакета **UTL_FILE** результаты **MY_DBMS_OUTPUT** можно сбросить в файл на клиенте точно так же, как и результаты пакета **DBMS_OUTPUT,** что делает пакет **MY_DBMS_OUTPUT** достойной заменой **DBMS_OUTPUT** для удаленного клиента.

Вы можете спросить, почему я использовал при реализации объектный тип, а не временную таблицу. Причина в объеме кода и дополнительном расходе ресурсов. Объем кода для управления временной таблицей, связанного с добавлением столбца для запоминания порядка данных, по сравнению с этой простой реализацией окажется существенно больше. Кроме того, работа с временной таблицей требует ввода-вывода и дополнительного расхода ресурсов. Наконец, сложно реализовать "представление со сбросом", когда буфер результатов автоматически очищается при выборке данных. Короче, использование объектного типа облегчает реализацию. Если бы я собирался использовать этот пакет для выдачи десятков мегабайт результатов, то пересмотрел бы способ буферизации и использовал временную таблицу. Для средних же объемов данных эта реализация вполне подходит.

Резюме

В этом разделе мы рассмотрели пакет **DBMS_OUTPUT.** Теперь, зная, как он работает, вы не пострадаете от побочных эффектов этой реализации. Вы будете готовы к тому, что между запрошенным размером буфера и суммарным объемом результатов, которые можно в него выдать, иногда нет очевидной зависимости. Вы будете знать, что выдать строку результатов длиной более 255 байт нельзя. Результаты **DBMS_OUTPUT** не выдаются, пока не **завершится** выполнение процедуры или оператора, но и тогда они будут выданы только при условии, что среда, из которой выполняются запросы, **поддерживает** пакет **DBMS_OUTPUT.**

Помимо анализа особенностей пакета **DBMS_OUTPUT** мы рассмотрели способы обойти ограничения, связанные с его применением: для этого рекомендуется использование других средств. Можно использовать пакет **UTL_FILE** для создания текстовых файлов с результатами или процедуры типа P, не только уменьшающие количество набираемых символов, но и обеспечивающие выдачу длинных строк. Можно реализовать и собственный пакет с аналогичными функциями, не имеющий подобных ограничений.

Пакет **DBMS_OUTPUT** — удачный пример того, как тривиальный на первый взгляд компонент может оказаться сложной программой с неожиданными побочными эффектами. Когда читаешь описание пакета **DBMS_OUTPUT** в руководстве Oracle *Supplied PL/SQL Packages Reference,* все кажется простым и понятным. А потом неожиданно возникают проблемы с суммарным объемом выдаваемых результатов и т.п. Знание особенностей реализации пакета помогает избежать этих проблем.

Пакет DBMS_PROFILER

Появления стандартного средства профилирования ждали давно (по крайней мере, я). Пакет **DBMS_PROFILER** представляет собой профилировщик исходного кода для PL/SQL-приложений. Раньше приходилось настраивать производительность PL/SQL-приложений с помощью средств **SQL_TRACE** и **TKPROF.** Они помогали выявить долго выполняющиеся SQL-операторы, но определить узкие места в в PL/SQL-коде из 5000 строк (особенно, если он написан кем-то другим) было практически невозможно. Чтобы определить проблемные фрагменты кода, приходилось вставлять в него множество вызовов функции **DBMS_UTILITY.GET_TIME** для измерения времени выполнения.

Теперь этого делать не нужно: можно воспользоваться возможностями пакета **DBMS_PROFILER.** Я собираюсь продемонстрировать, как его обычно используют. Лично я использую небольшую часть функциональных возможностей этого пакета: нахожу с его помощью проблемные фрагменты и занимаюсь ими непосредственно. Я использую пакет **DBMS_PROFILER** очень примитивным способом. Он позволяет, однако, делать намного больше, чем представлено в этом разделе.

Статистическая информация собирается в таблицы базы данных. Они позволяют сохранять статистическую информацию за несколько "прогонов" кода. Некоторых это устраивает, но я предпочитаю сохранять результаты одного-двух последних прогонов. Дополнительная информация лишь сбивает с толку. Иногда информации бывает слишком много.

Администратору базы данных, возможно, придется установить профилировщик в базе данных. Процедура установки этого пакета проста:

- **cd [ORACLE_HOME]/rdbms/admin;**

- с помощью **SVRMGRL** (или SQL*Plus — *прим. научн. ред.)* подключиться как **SYS** или **INTERNAL**;
- выполнить сценарий **profload.sql.**

После этого надо установить таблицы статистической информации. Их можно установить в базе данных в одном экземпляре, но я рекомендую каждому разработчику создать свой набор этих таблиц. К счастью, пакет **DBMS_PROFILER** создан с правами вызывающего и использует неуточненные имена таблиц, так что таблицы статистической информации можно установить в каждой схеме, и они будут корректно использоваться пакетом-профилировщиком. При использовании собственных таблиц каждый разработчик будет видеть только свои результаты профилирования, а не результаты коллег по работе. Чтобы создать таблицы статистической информации в своей схеме, надо выполнить сценарий **[ORACLE_HOME]\rdbms\admin\proftab.sql** в SQL*Plus. После выполнения сценария **proftab.sql** надо выполнить сценарий **profrep.sql**. Этот сценарий создает представления и пакеты для создания отчетов по таблицам профилировщика. Сценарий **profrep.sql** находится в файле **[ORACLE_HOME]\plsql\demo\profrep.sql**. Этот сценарий надо выполнить в своей схеме после создания таблиц.

Я обычно создаю небольшой сценарий для очистки таблиц профилировщика и выполняю эту очистку постоянно. После одного-двух тестовых прогонов и анализа результатов я выполняю этот сценарий. В сценарии, который я назвал **profreset.sql**, выполняется следующее:

— используются операторы delete, поскольку таблицы связаны требованием внешнего ключа

```
delete from plsql_profiler_data;
delete from plsql_profiler_units;
delete from plsql_profiler runs;
```

Теперь можно начинать профилирование. Я собираюсь продемонстрировать использование этого пакета путем запуска двух разных реализаций алгоритма вычисления факториала. Один алгоритм — рекурсивный, а другой — итеративный. Для того чтобы определить, какой из них работает быстрее и какие фрагменты кода наиболее медленные в каждой реализации, используется профилировщик. Тестирование выполняется следующим образом:

```
tkyte@TKYTE816> @profreset
tkyte@TKYTE816>create or replace
  2  function fact_recursive(n int) return number
  3   as
  4  begin
  5          if (n = 1)
  6          then
  7                  return 1;
  8          else
  9                  return n * fact_recursive(n-1);
 10          end if;
 11  end;
 12  /

Function created.
```

```
tkyte@TKYTE816> create or replace
  2  function fact_iterative(n int) return number
  3  as
  4          l_result number default 1;
  5  begin
  6         for i in 2 .. n
  7          loop
  8                 l_result := l_result * i;
  9          end loop;
 10          return l_result;
 11 end;
 12  /

Function created.

tkyte@TKYTE816> set serveroutput on

tkyte@TKYTE816> exec dbms_profiler.start_profiler('factorial recursive')

PL/SQL procedure successfully completed.

tkyte@TKYTE816> begin
  2         for i in 1 .. 50 loop
  3                 dbms_output.put_line(fact_recursive(50));
  4          end loop;
  5  end;
  6  /
30414093201713378043612608166064768844300000000000000000000000000

30414093201713378043612608166064768844300000000000000000000000000

PL/SQL procedure successfully completed.

tkyte@TKYTE816> exec dbms_profiler.stop_profiler

PL/SQL procedure successfully completed.

tkyte@TKYTE816> exec dbms_profiler.start_profiler('factorial iterative')

PL/SQL procedure successfully completed.

tkyte@TKYTE816> begin
  2         for i in 1 .. 50 loop
  3                 dbms_output.put_line(fact_iterative(50));
  4          end loop;
  5  end;
  6  /
30414093201713378043612608166064768844300000000000000000000000000

30414093201713378043612608166064768844300000000000000000000000000

PL/SQL procedure successfully completed.

tkyte@TKYTE816> exec dbms_profiler.stop_profiler

PL/SQL procedure successfully completed.
```

Для сбора статистической информации очередного прогона профилировщика необходимо вызвать процедуру **START_PROFILER**. Каждому прогону мы даем определен-

ное имя, а затем выполняем соответствующий код. Я вызывал каждую из функций вычисления факториала 50 раз, прежде чем завершить сбор статистической информации для прогона. Теперь все готово для анализа результатов.

В каталоге **[ORACLE_HOME]/plsql/demo** есть сценарий **profsum.sql**. Не запускайте его: некоторые запросы этого сценария выполняются очень долго (иногда — несколько часов), и он выдает очень много данных. (Ниже представлен измененный сценарий **profsum.sql**, который использую я; он выдает почти ту же информацию, но запросы выполняются очень быстро, а многие отчеты с избыточной детализацией просто не создаются.) Кроме того, одни запросы учитывают время выполнения вызова **STOP_PROFILER**, другие — нет. Это затрудняет сравнение результатов запросов. Я изменил все запросы так, чтобы время выполнения вызовов пакета профилировщика нигде не учитывалось.

Мой сценарий **profsum.sql** представлен ниже. Кроме того, он доступен для загрузки на сайте **http://www.wrox.com**:

```
set echo off
set linesize 5000
set trimspool on
set serveroutput on
set termout off

column owner format all
column unit_name format a14
column text format a21 word_wrapped
column runid format 9999
column secs  format 999.99
column hsecs format 999.99
column grand_total  format 9999.99
column run_comment format all word_wrapped
column line# format 99999
column pct format 999.9
column unit_owner format all

spool profsum.out

/* Очистка и пересоздание итоговых результатов. */
update plsql_profiler_units set total_time = 0;

execute prof_report_utilities.rollup_all_runs;

prompt =
prompt =
prompt ======================
prompt Суммарное время
select grand_total/1000000000 as grand_total
  from plsql_profiler_grand_total;

prompt =
prompt =
prompt======================
prompt Суммарное время каждого прогона
select runid,
       substr(run_comment,1, 30) as run_comment,
```

```
       run_total_time/1000000000 as secs
  from (select a.runid, sum(a.total_time)  run_total_time, b.run_comment
          from plsql_profiler_units a, plsql_profiler_runs b
         where a.runid = b.runid group by a.runid, b.run_comment)
 where run_total_time > 0
 order by runid asc;

prompt =
prompt =
prompt==================
prompt Процент времени, приходящийся на каждый модуль, отдельно для
-> каждого прогона
select pl.runid,
       substr(p2.run_comment, 1, 20) as run_comment,
       pl.unit_owner,
       decode(pl.unit_name, '', '<anonymous>',
                    substr(pl.unit_name,1, 20)) as unit_name,
       pl.total_time/1000000000 as secs,
       TO_CHAR(100*pl.total_time/p2.run_total_time, '999.9') as
percentage
  from plsql_profiler_units p1,
       (select a.runid, sum(a.total_time)  run_total_time, b.run_comment
          from plsql_profiler_units a, plsql_profiler_runs b
         where a.runid - b.runid group by a.runid, b.run_comment) p2
 where pl.runid=p2.runid
   and pl.total_time > 0
   and p2.run_total_time > 0
   and  (pl.total_time/p2.run_total_time)  >=  .01
 order by pl.runid asc, pl.total_time desc;

column secs form 9.99
prompt =
prompt =
prompt       -
prompt Процент времени, приходящийся на каждый модуль, суммарно по всем
-> прогонам
select pl.unit_owner,
       decode(pl.unit_name, '', '<anonymous>', substr(pl.unit_name,1, 25))
as unit_name,
       pl.total_time/1000000000 as secs,
       TO_CHAR(100*pl.total_time/p2.grand_total, '99999.99') as percentage
  from plsql_profiler_units_cross_run p1,
       plsql_profiler_grand_total p2
 order by pl.total_time DESC;

prompt =
prompt =
prompt========================
prompt Строки, потребовавшие более 1% суммарного времени, отдельно по
-> каждому прогону
select pl.runid as runid,
       pl.total_time/10000000 as Hsecs,
       pl.total_time/p4.grand_total*100 as pct,
```

```
       substr(p2.unit_ovmer, 1, 20) as owner,
       decode(p2.unit_name, '', '<anonymous>', substr(p2.unit_name,1, 20))
as unit_name,
       pl.line#,
        (select p3.text
           from all_source p3
          where p3.owner = p2.unit_owner and
                p3.line = pl.line# and
                p3.name=p2.unit_name and
                p3.type not in ('PACKAGE', 'TYPE')) text
  from plsql_profiler_data p1,
       plsql_profiler_units p2,
       plsql_profiler_grand_total p4
 where (pl.total_time >= p4.grand_total/100)
   AND pl.runID = p2.runid
   and p2.unit_number=p1.unit_number
 order by pl.total_time desc;
prompt =
prompt =
prompt=========================
prompt Наиболее популярные строки (более 1%), суммарно по всем прогонам
select pl.total_time/10000000 as hsecs,
        pl.total_time/p4.grand_total*100 as pct,
       substr(pl.unit_owner, 1, 20) as unit_owner,
       decode(pl.unit_name, '', '<anonymous>',
                 substr(pl.unit_name,1, 20)) as unit_name,
       pl.line#,
        (select p3.text from all_source p3
           where (p3.1ine = p1.line#) and
                 (p3.owner = pl.unit_owner) AND
                 (p3.name = pl.unit_name) and
                 (p3.type not in ('PACKAGE', 'TYPE'))) text
  from  plsql_profiler_lines_cross_run p1,
        plsql_profiler_grand_total p4
 where (pl.total_time >= p4.grand_total/100)
 order by pl.total_time desc;

execute prof_report_utilities.rollup_all_runs;

prompt =
prompt =
prompt  = = = = = = = = =
prompt Количество реально выполненных строк в программных единицах (с

-> группировкой по unit_name)

select pl.unit_owner,
       pl.unit_name,
       count(decode( pl.total_occur, 0, null, 0))  as lines_executed ,
       count(pl.line#) as lines_present,
       count(decode( pl.total_occur, 0, null, 0))/count(pl.line#) *100
                                        as pct
  from plsql_profiler_lines_cross_run p1
 where (pl.unit_type in ('PACKAGE BODY', 'TYPE BODY',
```

```
                              'PROCEDURE', 'FUNCTION'))
 group by pl.unit_owner, pl.unit_name;

prompt =
prompt =
prompt====================
prompt  Количество реально выполненных строк для всех программных единиц
select count(pl.line#) as lines_executed
  from plsql_profiler_lines_cross_run p1
 where (pl.unit_type in ('PACKAGE BODY', 'TYPE BODY',
                              'PROCEDURE', 'FUNCTION'))
    AND pl.total_occur > 0;

prompt =
prompt =
prompt       -
prompt  Общее количество строк во всех программных единицах
select count(pl.line#) as lines_present
  from plsql_profiler_lines_cross_run p1
 where (pl.unit_type in ('PACKAGE BODY', 'TYPE BODY',
                            'PROCEDURE','FUNCTION'));

spool off
set termout on
edit profsum.out
set linesize 131
```

Я постарался поместить отчет в стандартное окно терминала шириной 80 символов. Вы можете изменить формат некоторых столбцов, если не так часто используете программу Telnet.

Рассмотрим результаты, которые получены при тестировании функций, вычисляющих факториал, т.е. результаты работы представленного выше сценария **profsum.sql.**

```
Суммарное время

GRAND_TOTAL

       5.57
```

Суммарное время выполнения двух тестов составило 5,S7 секунды. Теперь посмотрим, сколько выполнялся каждый тест.

```
Суммарное время каждого прогона

RONID RUN_COMMENT    SECS

   17 factorial      3.26
      recursive

   18 factorial      2.31
      iterative
```

Рекурсивная версия уступает по производительности — она выполнялась почти в полтора раза дольше. Теперь посмотрим, сколько выполнялся каждый модуль (пакет или процедура) в тесте и какой процент это составляет от общего времени выполнения.

Процент времени, приходящийся на каждый модуль (отдельно для каждого прогона):

```
RUNID RUN_COMMENT UNIT_OWNER  UNIT_NAME        SECS PERCEN

   17 factorial   TKYTE      FACT_RECURSIVE    1.87   57.5
      recursive
   17 factorial   SYS        DBMS_OUTPUT       1.20   36.9
      recursive
   17 factorial   <anonymous> <anonymous>        .08    2.5
      recursive
   17 factorial   <anonymous> <anonymous>        .06    1.9
      recursive
   18 factorial   SYS        DBMS_OUTPUT       1.24   53.6
      iterative
   18 factorial   TKYTE      FACT_ITERATIVE     .89   38.5
      iterative
   18 factorial   <anonymous> <anonymous>        .08    3.4
      iterative
   18 factorial   <anonymous> <anonymous>        .06    2.7
      iterative

8 rows selected.
```

По этим данным видно, что в рекурсивной реализации 57% времени приходится на выполнение нашей функции, 37% — на выполнение процедуры **DBMS_OUTPUT,** а остальное время выполняются прочие подпрограммы. Во втором тесте результаты существенно отличаются. На выполнение нашего кода пришлось лишь 38% суммарного времени, причем это проценты от существенно меньшего времени! Это убедительно показывает, что вторая реализация эффективней первой. Столбец **SECS** содержит еще более показательные результаты. Как видите, рекурсивная функция выполнялась 1,87 секунды, а итеративная — 0,89. Если проигнорировать выполнение операторов **DBMS_OUTPUT,** окажется, что итеративная функция работает вдвое быстрее, чем рекурсивная.

Учтите, что результаты в вашей системе могут отличаться. Если не выполнить команду **SERVEROUTPUT ON** в SQL*Plus, например, вызовы **DBMS_OUTPUT** могут даже не попасть в отчет. Если выполнять тесты на других машинах, значения будут существенно отличаться. Например, при выполнении тестов на машине Sparc Solaris, суммарное время **(GRAND_TOTAL)** составило около 1,0 секунды, а время выполнения каждого раздела кода отличалось. В процентах, тем не менее, конечные результаты практически совпали.

Теперь рассмотрим, сколько времени суммарно в тестах выполнялся каждый модуль. Это покажет, какой фрагмент кода выполнялся дольше всего.

Процент времени, приходящийся на каждый модуль, суммарно по всем прогонам:

```
UNIT_OWNER  UNIT_NAME        SECS  PERCENTAG

SYS         DBMS_OUTPUT      2.44      43.82
TKYTE       FACT_RECURSIVE   1.87      33.61
TKYTE       FACT_ITERATIVE    .89      16.00
<anonymous> <anonymous>       .33       5.88
SYS         DBMS_PROFILER     .04        .69
```

Очевидно, что время выполнения можно уменьшить почти вдвое, убрав один вызов **DBMS_OUTPUT**. Если просто выполнить **SET SERVEROUTPUT OFF**, отключив выполнение **DBMS_OUTPUT**, и повторно выполнить тесты, окажется, что на эту процедуру приходится менее 3% общего времени выполнения. Сейчас, однако, именно эта процедура выполнялась дольше всего. Что еще интереснее — 33% времени заняло выполнение рекурсивной функции и 16% — итеративной. Итеративная функция работает намного быстрее.

Теперь рассмотрим более детальную информацию.

Строки, для выполнения которых потребовалось более 1% суммарного времени, -> отдельно по каждому прогону:

```
RUNID   HSECS    PCT OWNER UNIT_NAME      LINE TEXT

   17  142.47   25.6 TKYTE FACT_RECURSIVE:    8 return n*fact_recursive(n-1);
   18   68.00   12.2 TKYTE FACT_ITERATIVE     7 l_result := l_result * i;
   17   43.29    7.8 TKYTE FACT_RECURSIVE    4 if ( n = 1 )
   17   19.58    3.5 SYS   DBMS_OUTPUT     116 a3 a0 51 a5 lc 6e 81 b0
   18   19.29    3.5 TKYTE FACT_ITERATIVE    5 for i in 2 .. n
   18   17.66    3.2 SYS   DBMS_OUTPUT     116 a3 a0 51 a5 lc 6e 81 b0
   17   14.76    2.7 SYS   DBMS_OUTPUT     118 lc 51 81 b0 a3 a0 lc 51
   18   14.49    2.6 SYS   DBMS_OUTPUT     118 lc 51 81 b0 a3 a0 lc 51
   18   13.41    2.4 SYS   DBMS_OUTPUT     142 :2 a0 a5 b b4 2e d b7 19
   17   13.22    2.4 SYS   DBMS_OUTPUT     142 :2 a0 a5 b b4 2e d b7 19
   18   10.62    1.9 SYS   DBMS_OUTPUT      166 6e Ы 2e d :2 a0 7e 51 b4
   17   10.46    1.9 SYS   DBMS_OUTPUT     166 6e b4 2e d :2 a0 7e 51 b4
   17    8.11    1.5 SYS   DBMS_OUTPUT      72 1TO_CHAR:
   18    8.09    1.5 SYS   DBMS_OUTPUT     144 8f a0 b0 3d b4 55 6a :3 a0
   18    8.02    1.4 SYS   DBMS_OUTPUT      72 lTO_CHAR:
   17    8.00    1.4 SYS   DBMS_OUTPUT     144 8f a0 b0 3d b4 55 6a :3 a0
   17    7.52    1.4 <ano> <anonymous>       3
   18    7.22    1.3 <ano> <anonymous>       3
   18    6.65    1.2 SYS   DBMS_OUTPUT     141 a0 b0 3d b4 55 6a :3 a0 7e
   18    6.21    1.1 <ano> <anonymous>       1
   17    6.13    1.1 <ano> <anonymous>       1
   18    5.77    1.0 SYS   DBMS_OUTPUT      81 lORU-10028:: line length

22 rows selected.
```

Здесь выдается время выполнения (в сотых долях секунды) и процент от общего времени выполнения. В этих результатах нет ничего удивительного: можно было предположить, что дольше всего будет выполняться строка 8 рекурсивной и строка 7 итеративной функции. Это предположение подтверждается. В этой части кода показываются

конкретные строки кода, на которые надо обратить внимание. Обратите внимание на странного вида строки кода, начинающиеся с DBMS_OUTPUT. Так выглядит скрытый PL/SQL-код в базе данных. Это просто последовательность байтов, представляющая исходный код и скрывающая его от любопытных глаз.

В следующей части отчета представлены суммарные результаты по всем тестам, тогда как в предыдущей части проценты вычислялись для каждого теста отдельно.

```
Наиболее "популярные" строки (более 1%), суммарно по всем прогонам:

  HSECS    PCT OWNER UNIT_NAME      LINE TEXT
------------------------------------------------------------------------
 142.47   25.6 TKYTE FACT_RECURSIVE    8 return n * fact_recursive(n-1);
  68.00   12.2 TKYTE FACT_ITERATIVE    7 1 result := 1 result * i;
  43.29    7.8 TKYTE FACT_RECURSIVE    4 if ( n = 1 )
  37.24    6.7 SYS   DBMS OUTPUT     116 a3 a0 51 a5 lc 6e 81 b0
  29.26    5.3 SYS   DBMS OUTPUT     118 lc 51 81 b0 a3 a0 lc 51
  26.63    4.8 SYS   DBMS_OUTPUT     142 :2 a0 a5 b b4 2e d b7 19
  21.08    3.8 SYS   DBMS_OUTPUT     166 6e b4 2e d :2 a0 7e 51 b4
  19.29    3.5 TKYTE FACT_ITERATIVE    5 for i in 2 .. n
  16.88    3.0 <ano> <anonymous>       1
  16.13    2.9 SYS   DBMS_OUTPUT      72 1TO CHAR:
  16.09    2.9 SYS   DBMS_OUTPUT     144 8f a0 b0 3d b4 55 6a :3 a0
  14.74    2.6 <ano> <anonymous>     . 3
  11.28    2.0 SYS   DBMS OUTPUT      81 1ORU-10028:: line length overflow,
  10.17    1.8 SYS   DBMS_OUTPUT     147 4f 9a 8f a0 b0 3d b4 55
   9.52    1.7 SYS   DBMS_OUTPUT      73 1DATE:
   8.54    1.5 SYS   DBMS_OUTPUT     117 a3 a0 lc 51 81 b0 a3 a0
   7.36    1.3 SYS   DBMS_OUTPUT     141 a0 b0 3d b4 55 6a :3 a0 7e
   6.25    1.1 SYS   DBMS_OUTPUT      96 1WHILE:
   6.19    1.1 SYS   DBMS_OUTPUT      65 1499:
   5.77    1.0 SYS   DBMS_OUTPUT     145 7e a0 b4 2e d a0 57 b3

20 rows selected.
```

Наконец, рассмотрим статистическую информацию о частоте выполнения отдельных строк кода. Она пригодится не только при профилировании и настройке производительности, но и при тестировании. Эта часть отчета показывает, какие операторы в коде выполнялись и какой процент кода "покрыт" в ходе тестирования:

```
Количество реально выполненных строк в программных единицах (с
К группировкой по unit_name)

UNIT_OWNER  UNIT_NAME       LINES_EXECUTED LINES_PRESENT     PCT

SYS         DBMS_OUTPUT                 51            88    58.0
SYS         DBMS_PROFILER                9            62    14.5
TKYTE       FACT_ITERATIVE               4             4   100.0
TKYTE       FACT_RECURSIVE               3             3   100.0

==
==
======================================

Количество реально выполненных строк для всех программных единиц
```

```
LINES_EXECUTED

           67
```

```
Общее количество строк во всех программных единицах

LINES_PRESENT

          157
```

Из 88 операторов пакета **DBMS_OUTPUT выполнены 51. Интересно, как пакет DBMS_PROFILER** подсчитывает строки или операторы. Утверждается, что функция **FACT_ITERATIVE** содержит 4 строки кода, **но** если обратиться **к** исходному коду:

```
function fact_iterative(n int) return number
as
        l_result number default 1;
begin
        for i in 2 .. n
        loop
                l_result := l_result * i;
        end loop;
        return l_result;
end;
```

О каких четырех строках идет речь — непонятно. Пакет DBMS_PROFILER считает операторы, а не строки кода. Речь идет о следующих четырех операторах:

```
        l_result number default 1;

        for i in 2 .. n

            l_result := l_result * i;

        return l_result;
```

Остальные строки, хотя и необходимы для компиляции и выполнения кода, к выполняемому коду не относятся и поэтому операторами не считаются. Пакет **DBMS_PROFILER** можно использовать для того, чтобы определить количество операторов, выполняемых в коде и в тестах.

Проблемы

Единственная проблема, с которой я сталкивался при использовании пакета **DBMS_PROFILER**, связана с большим объемом генерируемых им данных и временем анализа этих данных.

Небольшой тест, который мы выполнили, сгенерировал более 500 строк статистической информации в таблице **PLSQL_PROFILER_DATA.** Эта таблица содержит одиннадцать числовых столбцов, так что она не очень "широкая", но растет быстро. При выполнении каждого оператора в таблицу добавляется строка. Надо контролировать пространство, занимаемое таблицей, периодически удаляя из нее строки. Обычно эта проблема несущественна, но я видел, как при тестировании сложных PL/SQL-процедур в таблицу записывались тысячи (и даже сотни тысяч) строк.

Время анализа результатов — более серьезная проблема. Сколько бы вы не настраивали производительность, **всегда** найдется строка кода, выполняющаяся дольше всего. Если удалить эту строку кода, ее место займет другая. Вы никогда не получите отчет **DBMS_PROFILER,** изучив который придете к выводу, что все работает прекрасно и настраивать больше нечего. Чтобы эффективно использовать это инструментальное средство, надо определить для себя, когда можно закончить настройку. Задайте либо определенное время настройки (например, эта процедура будет настраиваться в течение двух часов), либо критерий производительности (если процедура будет выполняться за N единиц времени, можно прекратить настройку). В противном случае вы будете (как и я иногда) тратить огромное время на настройку процедуры, которая просто не может работать быстрее.

Пакет **DBMS_PROFILER** — замечательное средство, которое может выдать массу подробной информации. Старайтесь не погрязнуть в изучении всех этих деталей.

Резюме

В этом разделе мы рассмотрели использование пакета **DBMS_PROFILER.** Он используется в основном для решения двух задач. При профилировании исходного кода можно найти строки, выполняющиеся дольше всего, или сравнить скорость работы двух алгоритмов. Можно также понять, какая часть кода охвачена при тестировании приложения. Хотя 100-процентный охват кода при тестировании не гарантирует его безошибочности, но приближает к ней.

Мы также разработали отчет, построенный на базе предлагаемого корпорацией Oracle примера отчета профилировщика. Этот отчет выдает основную информацию, необходимую для успешного использования пакета **DBMS_PROFILER.** Он избавляет от необходимости изучать лишние детали, предоставляя итоговую информацию о том, что происходило в приложении, и подробно описывая наиболее неэффективные части. Этого отчета может оказаться вполне достаточно для выявления узких мест и настройки приложения.

Пакет DBMS_UTILITY

Пакет **DBMS_UTILITY** — это набор процедур различного назначения. В него помещено несколько отдельных, не связанных между собой процедур. Пакет **DBMS_UTILITY** стандартно устанавливается в базе данных, и привилегия **EXECUTE** для него предоставляется роли **PUBLIC.** Процедуры в этом пакете не взаимосвязаны, как в большинстве остальных пакетов. Например, все подпрограммы пакета **UTL_FILE** имеют общее назначение — выполнение ввода-вывода в файл. Подпрограммы в пакете **DBMS_UTILITY** практически независимы.

Мы рассмотрим некоторые из этих подпрограмм, уделив особое внимание потенциальным проблемам при их использовании.

Процедура COMPILE_SCHEMA

Процедура **COMPILE_SCHEMA** предназначена для перекомпиляции *недействительных* (invalid) процедур, пакетов, триггеров, представлений, типов и других объектов схемы. Эта процедура работает в версии Oracle 8.1.6 на базе представления **SYS.ORDER_OBJECT_BY_DEPENDENCY.** Представление возвращает объекты в порядке зависимостей. Начиная с версии Oracle 8.1.7, это представление больше не используется (почему, будет показано далее). Если компилировать объекты в порядке, задаваемом этим представлением, объекты, которые **можно** успешно перекомпилировать, окажутся *действительными* (valid). Эта процедура выполняет оператор **ALTER COMPILE** от имени пользователя, который вызвал процедуру **COMPILE_SCHEMA** (т.е. она работает с правами вызывающего).

Процедура **COMPILE_SCHEMA** требует передавать имена пользователей в верхнем регистре. Если вызвать:

```
scott@TKYTE816> exec DBMS_UTILITY.compile_schema('scott');
```

скорее всего, ничего не произойдет, если при создании учетной записи имя пользователя scott не задано в нижнем регистре (как идентификатор в кавычках — *прим. научн.ред.)*. Необходимо передать имя схемы как SCOTT.

При использовании процедуры **COMPILE_SCHEMA** в версиях 8.1 сервера до 8.1.6.2 (т.е. во всех версиях 8.1.5, 8.1.6.0 и 8.1.6.1) возникает еще одна проблема. Если сервер поддерживает использование Java, в системе возникают рекурсивные зависимости. При вызове **COMPILE_SCHEMA** выдается сообщение об ошибке:

```
scott@TKYTE816> exec dbms_utility.compile_schema(user);
BEGIN dbms_utility.compile_schema(user); END;

*
ERROR at line 1:
ORA-01436: CONNECT BY loop in user data
ORA-06512: at "SYS.DBMS_UTILITY", line 195
ORA-06512: at line 1
```

Проблема связана с представлением **SYS.ORDER_OBJECT_BY_DEPENDENCY,** поэтому в версиях, начиная с Oracle 8.1.7, оно не используется. Если вы столкнетесь с этим сообщением об ошибке, можно создать собственную процедуру **COMPILE_SCHEMA,** работающую аналогично стандартной процедуре **COMPILE_SCHEMA.** В этой процедуре можно компилировать объекты в любом порядке. Типичное заблуждение состоит как раз в том, что объекты надо компилировать в строго определенном порядке. На самом деле компилировать объекты можно в произвольном порядке и получить тот же результат, что и при компиляции в порядке, определяемом зависимостями. Алгоритм следующий:

1. Выбираем недействительный объект схемы, который мы еще не пытались перекомпилировать.
2. Компилируем его.
3. Возвращаемся к первому шагу, пока есть недействительные объекты, которые мы еще не пытались перекомпилировать.

Определенного порядка придерживаться не обязательно. Причина — в побочном эффекте компиляции недействительного объекта. При этом все недействительные объекты, от которых он зависит, тоже будут скомпилированы. Надо только продолжать компилировать объекты, пока недействительных не останется. (На самом деле недействительные объекты могут остаться, но лишь потому, что скомпилировать их невозможно вообще.) Может оказаться, что при компиляции всего **одной** процедуры перекомпилированными окажутся 10 или 20 других объектов. Если не пытаться перекомпилировать эти 10 или 20 объектов вручную (при этом исходный объект снова станет недействительным), все будет в порядке.

Поскольку реализация этой процедуры представляет определенный интерес, я продемонстрирую ее. Для выполнения оператора **ALTER COMPILE** необходимо исполь-

зовать процедуру с правами вызывающего. Необходим также доступ к представлению **DBA_OBJECTS** для поиска следующего недействительного объекта и проверки состояния скомпилированного объекта. Не хотелось бы требовать обязательного доступа к представлению **DBA_OBJECTS** от пользователя, выполняющего процедуру. Для этого придется использовать подпрограммы как с правами вызывающего, так и с правами создателя. Необходимо, однако, сделать так, чтобы вызываемая пользователем основная процедура работала с правами вызывающего, — это обеспечит использование ролей.

Ниже представлена моя реализация процедуры **COMPILE_SCHEMA.**

*Пользователь, выполняющий этот сценарий, должен получить привилегию **SELECT** на представление **SYS.DBA_OBJECTS** непосредственно (подробнее об этом можно прочитать в главе 23).*

Поскольку это сценарий для утилиты SQL*Plus, включающий ряд директив SQL*Plus, я представлю только сам сценарий, а не результаты его выполнения. Для указания имени схемы при компиляции объектов я использую подставляемую переменную SQL*Plus. Это делается потому, что процедура выполняется с правами вызывающего (и если необходимо всегда обращаться к одной и той же таблице, независимо от того, кто выполняет процедуру, имя таблицы необходимо уточнять), а я лично предпочитаю не полагаться на общедоступные синонимы. Я представлю сценарий по частям, комментируя каждую часть:

```
column u new_val uname
select user u from dual;

drop table compile_schema_tmp
/

create  global temporary table compile_schema_tmp
(object_name varchar2(30),
 object_type varchar2(30),
 constraint compile_schema_tmp_pk
 primary key(object_name,object_type)
)
on commit preserve rows
/

grant all on compile_schema_tmp to public
/
```

Сценарий начинается с получения имени текущего зарегистрировавшегося пользователя в подставляемую переменную SQL*Plus. Она будет использоваться в операторах **CREATE OR REPLACE PROCEDURE**. Это приходится делать, поскольку процедура должна выполняться с правами вызывающего и обращаться к созданной выше таблице. В главе 23 было описано, как разрешаются ссылки на таблицы с использованием стандартной схемы для пользователя, выполняющего процедуру. Используется одна временная таблица для всех сеансов, которая будет принадлежать пользователю, выполнившему этот сценарий. Поэтому необходимо явно указать имя пользователя в PL/SQL-процедуре. Временная таблица используется процедурами для запоминания

того, какие объекты мы пытались перекомпилировать. Необходимо использовать конструкцию **ON COMMIT PRESERVE ROWS,** поскольку предполагается выполнять в процедурах операторы ЯОД (оператор **ALTER COMPILE** относится к операторам ЯОД), а после выполнения каждого такого оператора транзакция фиксируется. Теперь можно переходить к процедурам:

```
create or replace
procedure get_next_object_to_compile(p_username in varchar2,
                                     p_cmd out varchar2,
                                     p_obj out varchar2,
                                     p_typ out varchar2)
as
begin
    select 'alter ' || object_type || ' '
           || p_username || '.' || object_name ||
         decode(object_type, 'PACKAGE BODY', ' compile body',
                ' compile'),  object_name,  object_type
      into p_cmd, p_obj, p_typ
      from dba_objects a
     where owner = upper(p_username)
       and status = 'INVALID'
       and object_type <> 'UNDEFINED'
       and not exists (select null
                         from compile_schema_tmp b
                        where a.object_name = b.object_name
                          and a.object_type = b.object_type
                      )
       and rownum = 1;

    insert into compile_schema_tmp
    (object_name, object_type)
    values
    (p_obj, p_typ);
end;
/
```

Это процедура с правами создателя, с помощью которой мы будем обращаться к представлению **DBA_OBJECTS.** Она будет возвращать некий недействительный объект для перекомпиляции, если мы еще не пытались его компилировать. Процедура просто находит первый такой объект. По мере выбора мы запоминаем эти объекты во временной таблице. Процедура возбуждает исключительную ситуацию **NO_DATA_FOUND,** когда в запрошенной схеме не остается объектов, требующих перекомпиляции. Этот факт будет использоваться в следующей процедуре для прекращения обработки. Затем мы создадим процедуру с правами вызывающего, которая будет фактически выполнять компиляцию. Это объясняет, зачем в представленном выше коде понадобилась директива **COLUMN U NEW_VAL UNAME** — необходимо вставить имя **владельца** временной таблицы, чтобы избежать использования синонима. Поскольку мы делаем это динамически при компиляции процедуры, это решение лучше, чем использование синонима:

```
create or replace procedure compile_schema(p_username in varchar2)
authid current_user
```

```
as
    l_and  varchar2(512);
    l_obj dba_objects.object_name%type;
    l_typ dba_objects.object_type%type;
begin
    delete from &uname..compile_schema_tmp;

    loop
        get_next_object_to_compile(p_username, l_cmd, l_obj, l_typ);

        dbms_output.put_line(l_cmd);
        begin
            execute immediate l_cmd;
            dbms_output.put_line('Успешно');
        exception
            when others then
                dbms_output.put_line(sqlerrm);
        end;
        dbms_output.put_line(chr(9));
    end loop;

exception
    — процедура get_next_object_to_compile возбуждает эту
    — исключительную ситуацию, когда завершает работу
    when no_data_found then NULL;
end;
/

grant execute on compile_schema to public
/
```

Вот и все. Теперь можно переходить в любую схему, где есть компилируемые объекты, и выполнять:

```
scott@TKYTE816> exec tkyte.compile_schema('scott')
alter PROCEDURE scott.ANALYZE_MY_TABLES compile
Успешно

alter PROCEDURE scott.CUST_LIST compile
ORA-24344: success with compilation error

alter TYPE scott.EMP_MASTER compile
ORA-24344: success with compilation error

alter PROCEDURE scott.FOO compile
Успешно

alter PACKAGE scott.LOADLOBS compile
Успешно

alter PROCEDURE scott.P compile
Успешно

alter PROCEDURE scott.RUN_BY_JOBS compile
Успешно


PL/SQL procedure successfully completed.
```

Итак, процедуравыдает информацию о том, какие объекты она пытается компилировать, и результат компиляции. Судя по полученному результату, компилировалось семь объектов: при компиляции двух произошла ошибка, остальные пять успешно скомпилированы. Объекты компилировались в произвольном порядке — порядок просто не имеет значения. Эта процедура работает во всех версиях сервера.

Процедура ANALYZE_SCHEMA

Процедура **ANALYZE_SCHEMA** делает именно то, что можно предположить по ее названию, — выполняет операторы **ANALYZE** для сбора статистической информации об объектах в пользовательской схеме. Не рекомендуется применять ее для схем **SYS** или **SYSTEM**. В особенности не надо этого делать для схемы **SYS**, поскольку рекурсивные SQL-операторы, которые СУБД Oracle генерирует уже многие годы, оптимизированы для обработки оптимизатором, основанным на правилах. При наличии статистической информации о таблицах в схеме **SYS** сервер будет работать медленнее, чем мог бы. Эту процедуру можно использовать для анализа созданных пользователями прикладных схем.

Процедура **ANALYZE_SCHEMA** принимает пять аргументов.

- **SCHEMA**. Схема, которую необходимо проанализировать.
- **METHOD. ESTIMATE, COMPUTE** или **DELETE**. Если передано значение ESTIMATE, то одно из значений: **ESTIMATE_ROWS**, **ESTIMATE_PERCENT** должно быть ненулевым.
- **ESTIMATE_ROWS**. Количество оцениваемых строк.
- **ESTIMATE_PERCENT**. Процент оцениваемых строк. Если передано ненулевое значение параметра ESTIMATE_ROWS, этот параметр игнорируется.
- **METHOD_OPT [FOR TABLE] [FOR ALL [INDEXED] COLUMNS] [SIZE n] [FORALL INDEXES]**. Это те же опции, что используются в операторе ANALYZE. Они подробно описаны в руководстве *Oracle8i SQL Reference,* в разделе, посвященном конструкции FOR оператора **ANALYZE**.

Итак, например, все объекты в пользовательской схеме **SCOTT** можно проанализировать следующим образом. Начнем с удаления статистической информации, а затем соберем ее снова:

```
scott@TKYTE816> exec dbms_utility.analyze_schema(user, 'delete');

PL/SQL procedure successfully completed.

scott@TKYTE816>select table_name, num_rows, last_analyzed
  2  from user_tables;

TABLE_NAME                     NUM_ROWS LAST_ANAL


BONUS
CREATE$JAVA$LOB$TABLE
DEPT
```

```
12 rows selected.

scott@TKYTE816> exec dbms_utility.analyze_schema(user, 'compute');

PL/SQL procedure successfully completed.

scott@TKYTE816>select table_name, num_rows, last_analyzed
  2  from user_tables;

TABLE_NAME                     NUM_ROWS LAST_ANAL

BONUS                                 0 03-FEB-01
CREATE$JAVA$LOB$TABLE                58 03-FEB-01
DEPT                                  4 03-FEB-01


12 rows selected.
```

Этот простой пример показывает, что оператор ANALYZE COMPUTE выполняется — столбцы **NUM_ROWS** и **LAST_ANALYZED** получили значения.

Процедура **ANALYZE_SCHEMA** работает в соответствии со своим названием. Если необходимо анализировать объекты с разной степенью детализации, она не поможет. Процедура применяет один и тот же метод анализа ко всем типам объектов. Например, при эксплуатации большого хранилища данных, если необходимо использовать гистограммы по определенным столбцам или наборам столбцов только в некоторых таблицах, процедуру **ANALYZE_SCHEMA** применять нельзя. С помощью процедуры **ANALYZE_SCHEMA** можно либо получить гистограммы для всех столбцов, либо не получить их вообще — избирательно обрабатывать столбцы нельзя. Если анализ объектов выходит за рамки элементарного, процедура **ANALYZE_SCHEMA** не позволит его выполнить. Она подходит для небольших и средних (по объему обрабатываемых данных) приложений. Если необходимо обрабатывать большие объемы данных, имеет смысл распараллелить анализ или использовать разные опции анализа для различных таблиц. Этого процедура **ANALYZE_SCHEMA** не обеспечивает.

При использовании процедуры **ANALYZE_SCHEMA** следует учитывать следующие особенности. Первая связана с применением процедуры **ANALYZE_SCHEMA** к изменяющейся схеме. Вторая — с тем, какие объекты процедура **ANALYZE_SCHEMA** не анализирует. Рассмотрим их последовательно.

Применение процедуры ANALYZE_SCHEMA к изменяющейся схеме

Предположим, процедура **ANALYZE_SCHEMA** выполняется в схеме **SCOTT**. В эту схему добавлено несколько больших таблиц, поэтому их анализ требует определенного времени. В другом сеансе вы удаляете или добавляете ряд объектов в схеме **SCOTT**. Удаленный объект не был обработан процедурой **ANALYZE_SCHEMA.** Когда процедура попытается его проанализировать, будет выдано сбивающее с толку сообщение:

```
scott@TKYTE816> exec dbms_utility.analyze_schema(user, 'compute');
BEGIN dbms_utility.analyze_schema(user, 'compute'); END;
```

```
*
ERROR at line 1:
ORA-20000: You have insufficient privileges for an object in this schema.
ORA-06512: at "SYS.DBMS_UTILITY", line 258
ORA-06512: at line 1
```

Очевидно, что все необходимые привилегии есть — объект принадлежит вашей схеме. Ошибка связана с тем, что таблицы, которую процедура пытается анализировать, больше нет. Вместо того чтобы определить отсутствие таблицы, процедура предполагает, что таблица существует и пользователю просто не хватает привилегий, чтобы ее проанализировать. В этом случае можно сделать только следующее:

- повторно выполнить процедуру **ANALYZE_SCHEMA;**
- не удалять объекты по ходу выполнения процедуры **ANALYZE_SCHEMA.**

Следует также помнить, что, если объект добавлен в схему **после** начала выполнения процедуры **ANALYZE_SCHEMA,** он не будет проанализирован — процедура его не увидит. Это — не большая проблема, поскольку процедура ANALYZE_SCHEMA выполнится успешно.

Процедура ANALYZE_SCHEMA анализирует не все

В процедуре **ANALYZE_SCHEMA** есть нерешенная проблема. Она не анализирует таблицы, организованные по индексу, если в них используется дополнительный сегмент (подробнее о таблицах, организованных по индексу, и дополнительных сегментах см. в главе 6). Например, если выполнить следующий код:

```
scott@TKYTE816>drop table t;

Table dropped.

scott@TKYTE816>create table t (x int primary key, y date)
  2  organization index
  3  OVERFLOW TABLESPACE TOOLS
  4  /

Table created.

scott@TKYTE816>execute dbms_utility.analyze_schema('SCOTT', 'COMPOTE')

PL/SQL procedure successfully completed.

scott@TKYTE816>select table_name, num_rows, last_analyzed
  2    from user_tables
  3   where table_name = 'T';

TABLE_NAME                     NUM_ROWS LAST_ANAL

T
```

таблица T не будет проанализирована. Однако если не указывать конструкцию **OVERFLOW:**

```
scott@TKYTE816> drop table t;

Table dropped.

scott@TKYTE816> create table t  (x int primary key, y date)
  2  organization index
  3  /

Table created.

scott@TKYTE816> execute dbms_utility.analyze_schema('SCOTT', 'COMPUTE')

PL/SQL procedure successfully completed.

scott@TKYTE816> select table_name, num_rows, last_analyzed
  2    from user_tables
  3   where table_name = 'T';
TABLE_NAME                       NUM_ROWS LAST_ANAL

T                                       0 03-FEB-01
```

таблица анализируется. Это не означает, что конструкцию **OVERFLOW** для таблиц, организованных по индексу, задавать не надо — просто такие таблицы анализируются отдельно, вручную.

Процедура ANALYZE_DATABASE

Ей я посвящаю очень короткий раздел. **Не используйте эту процедуру.** Ее не имеет смысла использовать в базе данных любого размера. Кроме того, она имеет неприятный побочный эффект: анализирует словарь данных (объекты, принадлежащие пользователю SYS, никогда не нужно анализировать). Не используйте эту процедуру. Просто забудьте о ее существовании.

Функция FORMAT_ERROR_STACK

На первый взгляд эта функция кажется очень полезной, но на самом деле она бесполезна. Фактически **FORMAT_ERROR_STACK** — это просто менее функциональная реализация функции **SQLERRM** (SQL ERRor Message). Простой пример поможет вам понять, что я имею в виду:

```
scott@TKYTE816> create or replace procedure p1
  2  as
  3  begin
  4          raise program_error;
  5  end;
  6  /

Procedure created.

scott@TKYTE816> create or replace procedure p2
  2  as
  3  begin
  4         p1;
  5  end;
```

```
  6  /

Procedure created.

scott8TKYTE816> create or replace procedure p3
  2  as
  3  begin
  4          p2;
  5  end;
  6  /

Procedure created.

scott@TKYTE816> exec p3
BEGIN p3; END;
*
EBROR at line 1:
ORA-06501: PL/SQL: program error
ORA-06512: at "SCOTT.P1", line 4
ORA-06512: at "SCOTT.P2", line 4
ORA-06512: at "SCOTT.P3", line 4
ORA-06512: at line 1
```

В случае возникновения ошибки, если она не перехвачена обработчиком, выдается весь стек ошибок, который можно будет использовать в программе, использующей интерфейсы Pro*C, OCI, JDBC и т.п. Можно ожидать, что функция **DBMS_UTILITY.FORMAT_ERROR_STACK** будет возвращать подобную информацию. Оказывается, однако, что эту важную информацию она теряет:

```
scottQTKYTE816> create or replace procedure p3
  2  as
  3  begin
  4          p2;
  5  exception
  6          when others then
  7                dbms_output.put_line(dbms_utility.format_error_stack);
  8  end;
  9  /

Procedure created.

scott@TKYTE816> exec p3
ORA-06501: PL/SQL: program error

PL/SQL procedure successfully completed.
```

Как видите, при вызове функции **FORMAT_ERROR_STACK** информация стека ошибок потеряна! Функция возвращает ту же информацию, что и **SQLERRM**:

```
scott@TKYTE816> create or replace procedure p3
  2  as
  3  begin
  4          p2;
  5  exception
  6          when others then
  7                  dbms_output.put_line(sqlerrm);
```

```
  8  end;
  9  /
Procedure created.

scott@TKYTE816> exec p3
ORA-06501: PL/SQL: program error

PL/SQL procedure successfully completed.
```

Я утверждал, что функция **FORMAT_ERROR_STACK** обеспечивает меньше возможностей, чем **SQLERRM**. Дело в том, что функция **SQLERRM** может не только возвращать сообщение о текущей ошибке, но и сообщение о любой ошибке, код которой передан как параметр:

```
scott@TKYTE816> exec dbms_output.put_line(sqlerrm(-1));
ORA-00001: unique constraint (.) violated

PL/SQL procedure successfully completed.
```

К сожалению, сейчас нет способа получить весь стек ошибок в языке PL/SQL. Чтобы получить фактический номер строки кода, при выполнении которой произошла ошибка, приходится пропускать фатальные ошибки до вызывающей процедуры в клиентском приложении.

Функция FORMAT_CALL_STACK

К счастью, эта функция действительно очень полезна — не то, что функция FORMAT_ERROR_STACK. Она возвращает текущий стек вызовов. С ее помощью можно написать ряд полезных утилит, например **MY_CALLER** и **WHO_AM_I**. (Они выдают информацию о том, откуда вызвана текущая подпрограмма и как она называется — *прим. научн. ред.)* Эти утилиты вызывают рассматриваемую функцию чтобы определить, из какой строки какой подпрограммы вызваны. Такие сведения могут пригодиться при отладке и регистрации. Кроме того, процедуры могут работать по-разному, в зависимости от того, кто их вызвал и в какой среде.

Прежде чем представить код моих утилит **MY_CALLER** и **WHO_AM_I**, давайте посмотрим, какую информацию можно получить в стеке вызовов и какие результаты должны выдавать эти утилиты. Если использовать процедуры **P1**, **P2**, **P3** из представленного ранее примера и переписать процедуру **P1** следующим образом:

```
scott@TKYTE816> create or replace procedure p1
  2  as
  3      l_owner     varchar2(30);
  4      l_name      varchar2(30);
  5      l_lineno    number;
  6      l_type      varchar2(30);
  7  begin
  8      dbms_output.put_line('--------------------');
  9    dbms_output.put_line(dbms_utility.format_call_stack);
 10      dbms_output.put_line('--------------------');
 11    who_called_me(l_owner, l_name, l_lineno, l_type);
 12    dbms_output.put_line(l_type || ' ' ||
 13                         l_owner || '.' || l_name ||
```

```
 14             '('              || l_lineno ||  ')');
 15      dbms_output.put_line('————————————————————');
 16      dbms_output.put_line(who_am_i);
 17      dbms_output.put_line('————————————————————');
 18      raise program_error;
 19  end;
 20  /

Procedure created,
```

мы получим следующий результат:

```
scott@TKYTE816> exec p3

————PL/SQL Call Stack ————
  object      line  object
  handle    number  name
 2f191e0          9  procedure SCOTT.P1
 39f0a9c          4  procedure SCOTT.P2
 3aae318          4  procedure SCOTT.P3

 3a3461c          1  anonymous block

PROCEDURE SCOTT.P2(4)

SCOTT.Pl(16)

BEGIN p3; END;
*
ERROR at line 1:
ORA-06501: PL/SQL: program error
ORA-06512: at "SCOTT.P2", line 8
ORA-06512: at "SCOTT.P3", line 4
ORA-06512: at line 1
```

Итак, мы видим весь стек вызовов для процедуры **P1**. Показано, что процедура **P1** была вызвана процедурой **P2**, процедура **P2** была вызвана процедурой **P3**, которая, в свою очередь, была вызвана из анонимного блока. Кроме того, в программном коде можно выяснить, что процедура **P1** была вызвана в строке 4 процедуры **SCOTT.P2**. Наконец, можно выяснить, что выполняется сейчас процедура **SCOTT.P1.**

Теперь, разобравшись, как выглядит стек вызовов и что мы хотим получить, можно представить код, позволяющий это сделать:

```
tkyte@TKYTE816> create or replace function my_caller return varchar2
  2
  3  as
  4      owner       varchar2(30);
  5      name        varchar2(30);
  6      lineno      number;
  7      caller_t    varchar2(30);
  8      call_stack  varchar2(4096) default dbms_utility.format_call_stack;
  9      n           number;
 10      found_stack BOOLEAN default FALSE;
```

```
 11      line        varchar2(255);
 12      cnt         number := 0;
 13  begin
 14
 15      loop
 16         n := instr(call_stack, chr(10));
 17         exit when (cnt = 3 or n is NULL or n = 0);
 18
 19         line := substr(call_stack, 1, n-1);
 20         call_stack := substr(call_stack, n+1);
 21
 22          if (NOT found_stack) then
 23             if (line like ' %handle%number%name%') then
 24                 found_stack := TRUE;
 25              end if;
 26          else
 27             cnt := cnt + 1;
 28              — cnt = 1 — это я
 29             — cnt = 2 — это подпрограмма, которая меня вызвала
 30             — cnt = 3 — это подпрограмма, которая вызвала
-> подпрограмму, вызвавшую меня
 31             if (cnt = 3) then
 32                lineno := to_number(substr(line, 13, 6));
 33                line   := substr(line, 21);
 34                 if (line like 'pr%') then
 35                    n := length('procedure ' ) ;
 36                 elsif (line like 'fun%') then
 37                    n := length('function ' ) ;
 38                 elsif (line like 'package body%') then
 39                    n := length('package body ' ) ;
 40                 elsif (line like 'pack%') then
 41                    n := length('package ' ) ;
 42                 elsif (line like 'anonymous block%') then
 43                    n := length('anonymous block ' ) ;
 44                  else — must be a trigger
 45                     n := 0;
 46                 end if;
 47                 if (n <> 0) then
 48                    caller_t := ltrim(rtrim(upper(substr(line,1,
-> n-1))));
 49                    line := substr(line, n) ;
 50                 else
 51                     caller_t := 'TRIGGER';
 52                     line := ltrim(line);
 53                 end if;
 54                 n := instr(line, ' . ' ) ;
 55                   owner := ltrim(rtrim(substr(line, 1, n-1)));
 56                   name  := ltrim(rtrim(substr(line, n+1)));
 57             end if;
 58          end if;
 59      end loop;
 60     return owner || '.' || name;
```

```
 61  end;
 62  /

Function created.

tkyte@TKYTE816>create or replace function who_am_i return varchar2
  2  as
  3  begin
  4     return my_caller;
  5  end;
  6  /
Functioncreated.
```

При наличии этих процедур можно реализовать интересные решения. В частности, они могут использоваться в следующих случаях.

- Для проверки. Процедуры проверки могут регистрировать не только **пользователя,** выполнившего определенное действие, но и код, выполнивший это действие.
- Для отладки. Например, если снабдить код вызовами **DBMS_APPLICATION_INFO.SET_CLIENT_INFO(WHO_AM_I),** в другом сеансе можно выполнить запросы к представлению **V$SESSION,** чтобы определить, какой фрагмент кода сейчас выполняется. Подробнее о пакете **DBMS_APPLICATION_INFO** см. в начале приложения А.

Функция GET_TIME

Эта функция возвращает время, прошедшее с определенного момента, с точностью до сотых долей секунды. Функция **GET_TIME** не возвращает значение текущего времени, как можно было предположить по ее названию. Ее можно использовать для измерения периода времени между событиями. Обычно эта функция используется следующим образом:

```
scott@TKYTE816> declare
  2      l_start  number;
  3      n        number := 0;
  4  begin
  5
  6     l_start := dbms_utility.get_time;
  7
  8     for x in 1 .. 100000
  9      loop
 10         n := n+1;
 11      end loop;
 12
 13   dbms_output.put_line('   it took ' ||
 14    round((dbms_utility.get_time-l_start)/100,   2) ||
 15         ' seconds...');
 16  end;
 17  /
it took .12 seconds...

PL/SQLprocedure successfully completed.
```

Итак, функция **GET_TIME** используется для измерения времени с точностью до сотых долей секунды. Нужно, однако, учитывать, что счетчик **GET_TIME** может переполниться, сбросится в **ноль** и начать отсчет сначала, если сервер работает достаточно долго. Сейчас на большинстве платформ сброс произойдет не раньше, чем через год. Для счетчика используется 32-битовое целое число, что позволяет хранить сотые доли секунды примерно для 497 дней. После этого произойдет переполнение счетчика, и отсчет начнется с нуля. На некоторых платформах операционная система увеличивает значение счетчика чаще, чем раз в одну сотую секунды. На этих платформах сброс счетчика может произойти раньше, чем через 497 дней. Например, на платформе Sequent таймер обнуляется за 71,58 минуты, поскольку в этой операционной системе счетчик отсчитывает микросекунды, поэтому 32-битовое целое переполняется существенно быстрее. На 64-битовых платформах счетчик не переполнится и за тысячи лет.

Последнее замечание о функции **GET_TIME.** Значение, возвращаемое функцией **GET_TIME,** может быть получено и с помощью оператора **SELECT * FROM V$TIMER.** Это динамическое представление и функция **GET_TIME** возвращают одно и то же:

```
tkyte@TKYTE816>selecthsecs,dbms_utility.get_time
  2  from v$timer;

     HSECS   GET_TIME

   7944822    7944822
```

Функция GET_PARAMETER_VALUE

Эта функция позволяет пользователю получить значение указанного параметра конфигурации. Даже при отсутствии доступа к представлению **V$PARAMETER** и невозможности выполнить команду **SHOW PARAMETER**, с помощью этой функции можно получить значение параметра инициализации сервера. Функция используется следующим образом:

```
scott@TKYTE816>show parameter utl_file_dir
ORA-00942: table or view does not exist

scott@TKYTE816>select * from v$parameter where name = 'utl_file_dir'
  2  /
select * from v$parameter where name = 'utl_file_dir'
              *
ERROR at line 1:
ORA-00942: table or view does not exist

scott@TKYTE816> declare
  2     intval number;
  3     strval varchar2(512);
  4  begin
  5    if (dbms_utility.get_parameter_value('utl_file_dir',
  6                                          intval,
  7                                         strval) = 0)
  8     then
  9     dbms_output.put_line('Значение=      ' || intval);
```

```
 10      else
 11         dbms_output.put_line( 'Значение = ' || strval);
 12      end if;
 13  end;
 14  /
Значение = c:\temp\

PL/SQL procedure successfully completed.
```

Как видите, хотя пользователь **SCOTT** не может обратиться к представлению **V$PARAMETER** и выполнить команду **SHOW PARAMETER**, он может получить требуемое значение. Следует отметить, что параметры инициализации со значениями True/False в файле init.ora будут выданы как числа: значение 1 обозначает **ИСТИНУ** (True), а значение 0 — ЛОЖЬ (False). Для параметров с несколькими значениями, например **UTL_FILE_DIR**, функция выдает только первое значение. Если использовать учетную запись, имеющую право для этой же базы данных выполнять команду **SHOW PARAMETER**:

```
tkyte@TKYTE816> show parameter utl_file_dir

NAME                                 TYPE    VALUE

utl_file_dir                         string  c:\temp, c:\oracle
```

можно получить все значения.

Процедура NAME_RESOLVE

Этой процедуре можно передать имя:

- процедуры верхнего уровня;
- функции верхнего уровня;
- пакета;
- синонима пакета, процедуры или функции верхнего уровня.

Она выдает полное имя соответствующего объекта.

Процедура позволяет узнать, является ли объект с указанным именем процедурой, функцией или пакетом и какой схеме он принадлежит. Рассмотрим простой пример:

```
scott@TKYTE816> declare
  2     type vcArray is table of varchar2(30);
  3     l_types vcArray := vcArray(null, null, null, null, 'synonym',
  4                                null, 'procedure', 'function',
  5                                'package');
  6
  7       l_schema   varchar2(30);
  8      l_part1     varchar2(30);
  9       l_part2    varchar2(30);
 10       l_dblink   varchar2(30);
 11        l_type     number;
 12        l_obj#     number;
 13  begin
```

```
 14  dbms_utility.name_resolve(name => 'DBMS_UTILITY',
 15                            context       => 1,
 16                            schema        => l_schema,
 17                  part1                   => l_part1,
 18                            part2         => l_part2,
 19                            dblink        => l_dblink,
 20                part1_type                => l_type,
 21                             object_number => l_obj#);
 22   if l_obj# IS NULL
 23   then
 24     dbms_output.put_line('Object not found or not valid.');
 25   else
 26   dbms_output.put(l_schema || '.' || nvl(l_part1,l_part2));
 27    if l_part2 is not null and l_part1 is not null
 28    then
 29       dbms_output.put('.' || l_part2);
 30    end if;
 31
 32    dbms_output.put_line(' is a ' || l_types(l_type) ||
 33                         ' with object id ' || l_obj# ||
 34             '             and dblink "' || l_dblink || '"');
 35   end if;
 36 end;
 37 /
SYS.DBMS_UTILITY is a package with object id 2408 and dblink ""

PL/SQL procedure successfully completed.
```

Процедура **NAME_RESOLVE** по синониму **DBMS_UTILITY** определила, что речь идет о пакете, принадлежащем пользователю **SYS**.

Следует отметить, что процедура **NAME_RESOLVE** работает только для процедур, функций, пакетов и синонимов, ссылающихся на один из этих трех типов объектов. В частности, она не работает для таблиц. При попытке передать имя таблицы EMP в схеме пользователя **SCOTT**, например, вы получите следующее сообщение об ошибке:

```
declare
*
ERROR at line 1:
ORA-06564: object emp does not exist
ORA-06512: at "SYS.DBMS_UTILITY", line 68
ORA-06512: at line 9
```

Помимо того, что процедура **NAME_RESOLVE** не работает с таблицами, индексами и другими объектами, она работает не так, как указано в документации, при разрешении синонимов, ссылающихся на удаленные объекты по связи базы данных. В документации написано, что если передать процедуре **NAME_RESOLVE** синоним удаленного пакета или удаленной процедуры, то в качестве типа объекта будет возвращен синоним, а в качестве имени — имя связи базы данных. Проблема — в коде процедуры **NAME_RESOLVE** (документация описывает предполагаемые результаты, но процедура не работает должным образом). В текущей реализации процедура **NAME_RESOLVE** никогда не возвращает **SYNONYM** в качестве типа объекта. Вместо этого она уточняет

имя удаленного объекта и возвращает его имя и идентификатор -1. Например, я настроил связь базы данных и создал синоним **X** для пакета **DBMS_UTILITY@ora8i.world.** При попытке уточнения имени этого синонима я получаю:

```
SYS.DBMS_UTILITY is a package with object id -1 and dblink ""

PL/SQL procedure successfully completed.
```

В выходном параметре **DBLINK** я должен был бы получить сообщение о том, что объект X — синоним и соответствующую информацию о связи. Как видите, однако, в переменной **DBLINK** мы получили значение Null, и единственный признак того, что пакет не является локальным, — значение -1 для идентификатора объекта. Не стоит предполагать, что эта особенность останется в следующих версиях СУБД Oracle. Было доказано, что проблема — в реализации процедуры **NAME_RESOLV**E, а не в документации. Документация правильно задает требования, но они некорректно реализованы. После исправления ошибки процедура **NAME_RESOLVE** будет работать для удаленных объектов иначе. Поэтому следует либо избегать использования процедуры **NAME_RESOLVE** для удаленных объектов, либо поместить вызов **NAME_RESOLVE** в подпрограмму, решающую эту проблему. Когда реализация этой процедуры изменится, можно будет реализовать прежние особенности работы для фрагментов кода, которые от них зависят.

Последнее замечание относительно процедуры **NAME_RESOLVE**. Параметр **CONTEXT** плохо описан, а параметр **OBJECT_NUMBER** не описан в документации вовсе. Про параметр **CONTEXT** написано только, что:

```
... должен быть целым числом от 0 до 8
```

На самом деле, его значение должно быть целым числом от 1 до 7, или будет получено сообщение об ошибке:

```
declare
*
ERROR at line 1:
ORA-20005: ORU-10034: context argument must be 1 or 2 or 3 or 4 or 5 or
6 or 7
ORA-06512: at "SYS.DBMS_UTILITY", line 66
ORA-06512: at line 14
```

А если задать любое значение из этого диапазона, кроме 1, будет выдано одно из следующих сообщений об ошибке:

```
ORA-04047: object specified is incompatible with the flag specified

ORA-06564: object OBJECT-NAME does not exist
```

Так что единственно допустимым значением контекста является 1. Параметр **OBJECT_NUMBER** вообще не описан. Это идентификатор объекта (значение столбца **OBJECT_ID**), который можно найти в представлениях **DBA_OBJECTS, ALL_OBJECTS** и **USER_OBJECTS**. Например, возвращаясь к первому примеру, где было выдано значение **OBJECT_ID** 2048, можно выполнить запрос:

```
scott@TKYTE816>select owner, object_name
  2  from all_objects
  3  where object_id = 2408;
```

```
OWNER                          OBJECT_NAME

SYS                            DBMS_UTILITY
```

Процедура NAME_TOKENIZE

Эта процедура разбивает строку, представляющую имя объекта, на компоненты. Для ссылок на объекты используется следующий синтаксис:

[схема].[имя_объекта].[процедура|функция]@[связь_баэы_данных]

Процедура NAME_TOKENIZE в строке такого вида выделяет три начальных компонента и связь базы данных. Кроме того, она сообщает, на каком байте от начала строки ею завершен анализ имени объекта. Ниже представлен простой пример, демонстрирующий, что можно получить при передаче процедуре имени объекта. Учтите, что имена **реальных** объектов передавать необязательно (соответствующие таблицы и процедуры могут и не существовать), но обязательно передавать **допустимые** имена объектов. Если передан недопустимый (синтаксически) идентификатор, процедура **NAME_TOKENIZE** возбудит исключительную ситуацию. Это позволяет проверить с помощью процедуры **NAME_TOKENIZE,** является ли строка символов допустимым идентификатором:

```
scott@TKYTE816> declare
  2      l_a      varchar2(30);
  3      l_b      varchar2(30);
  4      l_c      varchar2(30);
  5     l_dblink varchar2(30);
  6      l_next   number;
  7
  8     type vcArray is table of varchar2(255);
  9      l_names vcArray :=
 10        vcArray('owner.pkg.proc@database_link',
 11                'owner.tbl@database_link',
 12                 'tbl',
 13                '"Owner".tbl',
 14                 'pkg.proc',
 15                'owner.pkg.proc',
 16                 'proc',
 17          'owner.pkg.proc@dblink         with junk',
 18                 '123');
 19 begin
 20     for i in 1 .. l_names.count
 21      loop
 22      begin
 23          dbms_utility.name_tokenize(name   => l_names(i),
 24                                     a      => l_a,
 25                                     b      => l_b,
 26                                     c      => l_c,
 27                                     dblink => l_dblink,
 28                                     nextpos=> l_next);
 29
 30       dbms_output.put_line('name       ' || l_names(i));
```

```
 31          dbms_output.put_line('A      ' II l_a);
 32          dbms_output.put_line('B      ' II l_b);
 33          dbms_output.put_line('C      ' II l_c);
 34         dbms_output.put_line('dblink  ' || l_dblink);
 35        dbms_output.put_line('next    ' || l_next || ' ' ||
 36                                length(l_names(i)));
 37          dbms_output.put_line('————————————————————————') ;
 38      exception
 39          when others then
 40              dbms_output.put_line('name    ' || l_names(i));
 41              dbms_output.put_line(sqlerrm) ;
 42      end;
 43      end loop;
 44  end;
 45  /
name    owner.pkg.prod@database_link
A       OWNER
B       PKG
C       PROC
dblink  DATABASE_LINK
next    28 28
```

Как видите, процедура позволяет выделить компоненты имени объекта. Параметр NEXT получил значение длины строки — в данном случае анализ закончился, когда был достигнут конец строки. Поскольку мы задали полное имя объекта, все четыре компонента имени получили значения. Теперь рассмотрим другие примеры:

```
name   owner.tbl@database_link
A      OWNER
B      TBL
C
dblink DATABASE_LINK
next   23 23

name   tbl
A      TBL
B
C
dblink
next   3 3
```

Обратите внимание, что параметры B и C остались пустыми. Хотя идентификатор объекта строится по принципу **СХЕМА.ОБЪЕКТ.ПРОЦЕДУРА**, процедура **NAME_TOKENIZE** не поместила значение **TBL** в выходной параметр **B**. Она просто помещает первую часть имени в параметр **A**, следующую часть — в B и т.д. Параметры **A**, **B** и **C** не содержат логический компонент имени — они содержат первую найденную часть, вторую и т.д.

```
name   "Owner    ". t b l
A        Owner
B        TBL
```

```
C
dblink
next    11 11
```

Этот результат представляет определенный интерес. В предыдущих примерах процедура **NAME_TOKENIZE** выдавала все имена в верхнем регистре. Дело в том, что все идентификаторы сервер переводит в верхний регистр, если только они **не взяты в кавычки.** Мы же передали идентификатор в кавычках. В таком идентификаторе процедура **NAME_TOKENIZE** сохранит регистр символов, но кавычки удалит.

```
name    pkg.proc
A       PKG
B       PROC
C
dblink
next    8 8
-----------------------------
name    owner.pkg.proc
A       OWNER
B       PKG
C       PROC
dblink
next    14 14
-----------------------------
name    proc
A       PROC
B
C
dblink
next    4 4
-----------------------------
name  owner.pkg.proc@dblink  with junk
A       OWNER
B       PKG
C       PROC
dblink  DBLINK
next    22 31
-----------------------------
```

В этом примере анализ прекратился **раньше,** чем был достигнут конец строки. Процедура **NAME_TOKENIZE** сообщает, что прекратила анализ на 22-ом байте из 31. Это пробел перед строкой **with junk.** Процедура проигнорировала остаток строки после имени.

```
name    123
ORA-00931: missing identifier

PL/SQL procedure successfully completed.
```

Последний пример показывает, что при получении недопустимого идентификатора процедура **NAME_TOKENIZE** возбуждает исключительную ситуацию. Она проверяет, является ли каждая возвращаемая лексема допустимым идентификатором. Это позво-

ляет использовать ее для проверки допустимости имен в приложениях, создающих объекты в базе данных Oracle. Например, если создается средство построения модели данных и необходимо проверить допустимость имени, которое пользователь хочет задать для таблицы или столбца, можно применить процедуру **NAME_TOKENIZE.**

Процедуры COMMA_TO_TABLE, TABLE_TO_COMMA

Эти две процедуры преобразуют, соответственно, список **идентификаторов** через запятую в PL/SQL-таблицу **(COMMA_TO_TABLE)** и PL/SQL-таблицу произвольных строк в строку-список через запятую **(TABLE_TO_COMMA). Я** выделил слово **"идентификаторов",** потому что процедура **COMMA_TO_TABLE** использует для анализа строк процедуру NAME_TOKENIZE. Поэтому необходимо задавать допустимые идентификаторы Oracle (или идентификаторы в кавычках). При этом, однако, элемент списка все равно не может быть длиннее 30 символов.

Эта утилита наиболее полезна для приложений, которые хранят список, например, имен таблиц в одной строке, и преобразуют их по ходу работы в PL/SQL-таблицу. Применить эту таблицу для других целей не удастся. Универсальная процедура **COMMA_TO_TABLE,** работающая со строками данных, перечисленных через запятую, была представлена в главе 20. Там я демонстрирую на ее примере, как выбирать данные с помощью **SELECT** из PL/SQL-функции.

Вот пример использования стандартной процедуры **COMMA_TO_TABLE,** показывающий, как она обрабатывает длинные и недопустимые идентификаторы:

```
scott@TKYTE816> declare
  2    type vcArray is table of varchar2(4000);
  3
  4    l_names  vcArray :=vcArray('emp,dept,bonus',
  5                               'a,  b  ,   c',
  6                              423, 456, 789',
  7                               '"123", "456", "789"',
  8       "'This is a long string, longer then 32 characters","b",c');
  9     l_tablen number;
 10     l_tab   dbms_utility.uncl_array;
 11 begin
 12    for i in 1 .. l_names.count
 13     loop
 14       dbms_output.put_line(chr(10) ||
 15         '['          ||          l_names(i)  ||  ']');
 16     begin
 17
 18         dbms_utility.comma_to_table(l_names(i),
 19                                     l_tablen, l_tab);
 20
 21        for j in 1..l_tablen
 22         loop
 23         dbms_output.put_line('[' || l_tab(j) || ']');
 24         end loop;
```

```
25
26          l_names(i)  := null;
27          dbms_utility.table_to_comma(l_tab,
28                                      l_tablen, l_names(i));
29           dbms_output.put_line(l_names(i));
30       exception
31           when others then
32              dbms_output.put_line(sqlerrm) ;
33       end;
34        end loop;
35 end;
36 /

[emp,dept,bonus]
[emp]
[dept]
[bonus]
emp,dept,bonus
```

Этот **пример** демонстрирует как по строке **emp,dept,bonus** строится таблица идентификаторов, которая затем снова преобразуется в строку.

```
[a,    b,      c]
[a]
[   b   ]
[      c]
a,    b,      c
```

Этот пример показывает, что пробелы в списке сохраняются. Для удаления начальных и конечных пробелов, если они не нужны, надо использовать встроенную функцию **TRIM.**

```
[123, 456, 789]
ORA-00931: missing identifier
```

Это показывает, что для применения процедуры к строке чисел необходимо изменить формат строки, как показано ниже:

```
t"123",  "456",  "789"]
["123"]
[  "456"]
[  "789"]
"123",  "456",  "789"
```

Числа в кавычках удалось извлечь из строки. Обратите внимание, однако, что в элементах таблицы сохранен не только начальный пробел, но и кавычки. Если они не нужны, удалять их придется отдельно.

```
["This is a long string, longer than 32 characters","b",c]
ORA-00972: identifier is too long

PL/SQL procedure successfully completed.
```

Последний пример показывает, что, если передан слишком длинный идентификатор (длиннее 30 символов), процедура возбуждает исключительную ситуацию. Рассматриваемые процедуры подходят только для строк длиной до 30 символов. Хотя процеду-

ра **TABLE_TO_COMMA** и позволяет формировать строку из элементов таблицы длиннее 30 символов, процедура **COMMA_TO_TABLE** не позволит преобразовать результат обратно в таблицу.

Процедура DB_VERSION и функция PORT_STRING

Процедура **DB_VERSION** появилась в версии Oracle 8.0 для определения версии сервера в приложениях. Ее можно было бы использовать, например, в пакете **CRYPT_PKG** (см. раздел приложения А, посвященный пакету **DBMS_OBFUSCATION_TOOLKIT**), чтобы предупредить пользователей, пытающихся использовать для шифрования процедуры **DES3** на сервере Oracle 8.1.5, что это не сработает, до того, как они начнут получать сообщения об ошибках. Интерфейс этой процедуры предельно прост:

```
scott@TKYTE816> declare
  2      l_version        varchar2(255);
  3      l_compatibility varchar2(255);
  4  begin
  5      dbms_utility.db_version(l_version, l_compatibility);
  6      dbms_output.put_line(l_version);
  7      dbms_output.put_line(l_compatibility);
  8  end;
  9  /
8.1.6.0.0
8.1.6

PL/SQL procedure successfully completed.
```

При этом она выдает более детальную информацию, чем более старая функция **PORT_STRING:**

```
scott@TKYTE816>  select dbms_utility.port_string from dual;

PORT_STRING

----------------------------------
IBMPC/WIN_NT-8.1.0
```

Функция **PORT_STRING** не только вынуждает анализировать полученную строку, но и не позволяет определить, работаем ли мы с сервером версии 8.1.5, 8.1.6 или 8.1.7. Процедура **DB_VERSION** для получения такой информации подходит больше. С другой стороны, функция **PORT_STRING** позволяет определить, на какой операционной системе работает сервер.

Функция GET_HASH_VALUE

Эта функция возвращает для переданной строки числовое хеш-значение. Ее можно использования для создания собственной "PL/SQL-таблицы" со строковыми индексами или, как было показано в разделе, посвященном пакету **DBMS_LOCK,** для реализации других алгоритмов.

Учтите, что алгоритм, используемый для реализации функции **GET_HASH_VALUE,** менялся при переходе к следующей версии сервера, так что не надо использовать эту

функцию для генерации суррогатных ключей. Если сохранять возвращаемые этой функцией значения в таблице, при переносе приложения в следующую версию сервера возможны проблемы, поскольку по тем же входным данным будут выдаваться другие значения хеш-функции!

Функция принимает три параметра.

- Строку, которую необходимо хешировать.
- "Базовое число" для возвращаемых значений. Если необходимо получить числа в диапазоне от 0 до некоторого числа, надо указать базовое значение 0.
- Размер хеш-таблицы. Лучше, если это число будет степенью двойки.

Чтобы продемонстрировать использование функции **GET_HASH_VALUE**, я реализую новый тип, **HASHTABLETYPE,** для поддержки хешей в языке PL/SQL. Он очень похож на PL/SQL-таблицу, проиндексированную не числами, а строками типа **VARCHAR2.** Обычно элементы PL/SQL-таблицы индексируются целыми числами. Новый же тип PL/SQL-таблицы позволяет индексировать элементы строками. Можно объявлять переменные типа **HASHTABLETYPE,** помещать **(GET)** и выбирать **(PUT)** из них данные. Такого рода таблиц можно создать сколько угодно. Вот спецификация соответствующих типов:

```
tkyte@TKYTE816>create or replace type myScalarType
  2  as object
  3   (key  varchar2(4000),
  4   val  varchar2(4000)
  5  );
  6  /

Type created.

tkyte@TKYTE816>create or replace type myArrayType
  2  as varray(10000) of myScalarType;
  3  /

Type created.

tkyte@TKYTE816>create or replace type hashTableType
  2  as object
  3  (
  4     g_hash_size     number,
  5     g_hash_table    myArrayType,
  6     g_collision_cnt number,
  7
  8     static function new(p_hash_size in number)
  9         return hashTableType,
 10
 11     member procedure put(p_key in varchar2,
 12                          p_val  in varchar2),
 13
 14     member function get(p_key in varchar2)
 15         return varchar2,
 16
 17     member procedure print_stats
```

```
18  );
19  /
```

```
Type created.
```

Интересная особенность реализации состоит в добавлении статической функции-члена **NEW**. Это позволит создать собственный конструктор. Специального значения имя **NEW** не имеет. Это не ключевое слово. Функция **NEW** просто позволяет объявлять данные типа **HASHTABLETYPE** следующим образом:

```
declare
    l_hashTable hashTableType := hashTableType.new(1024);
```

а не как обычно:

```
declare
   l_hashTablehashTableType:=hashTableType(1024,myArrayType(),0);
```

Я уверен, что первый вариант надежнее и понятнее второго. Второй вариант вызова раскрывает многие детали реализации (например, то, что используется тип массива, имеется переменная **G_COLLISION_CNT**, которой надо задавать значение 0, и т.п.). Пользователям знать об этом не обязательно.

Теперь рассмотрим тело объектного типа:

```
scott@TKYTE816>create or replace type body hashTableType
  2  as
  3
  4  — Наш более "дружественный" конструктор.
  5
  6  static function new(p_hash_size in number)
  7  return hashTableType
  8  is
  9  begin
 10     return hashTableType(p_hash_size, myArrayType(), 0);
 11  end;
 12
 13  member procedure put(p_key in varchar2, p_val in varchar2)
 14  is
 15     l_hash  number :=
 16             dbms_utility.get_hash_value(p_key, 1, g_hash_size);
 17  begin
 18
 19     if (p_key is null)
 20     then
 21     raise_application_error(-20001, 'Пустой ключ не допускается');
 22     end if;
 23
```

Следующий фрагмент кода определяет, надо ли увеличить таблицу для размещения нового хешированного значения. Если — да, таблица увеличивается до необходимого для добавления индекса размера:

```
27       if (l_hash > nvl( g_hash_table.count, 0))
28       then
```

```
29            g_hash_table.extend(l_hash-nvl(g_hash_table.count,0)+1);
30        end if;
31
```

Нет никакой гарантии, что запись с индексом, полученным при хешировании соответствующего ключа, пуста. При выявлении конфликта мы пытаемся поместить значение в следующий элемент набора. Выполняется до 1000 попыток поместить значение в таблицу. Если все 1000 попыток завершатся неудачно, попытка добавления считается неудачной. Это значит, что размер таблицы недостаточен:

```
35    for i in 0 .. 1000
36     loop
37        — Если мы собираемся выйти за пределы таблицы,
38        — сначала надо добавить новый слот.
39        if (g_hash_table.count <= l_hash+i)
40         then
41            g_hash_table.extend;
42        end if;
43
```

Следующий фрагмент реализует действие: если слот не используется или ключ уже находится в этом слоте, использовать и вернуть его. Странной кажется проверка того, пуст ли элемент **G_HASH_TABLE** или значение **G_HASH_TABLE(L_HASH+I).KEY**. Это показывает, что элемент набора может быть пустым (Null) или может содержать объект с пустыми атрибутами:

```
46        if (g_hash_table(l_hash+i) is null OR
47            nvl(g_hash_table(l_hash+i).key,p_key) = p_key)
48         then
49          g_hash_table(l_hash+i) := myScalarType(p_key,p_val);
50             return;
51        end if;
52
53        — Иначе увеличить счетчик количества конфликтов
54         — и перейти к следующему слоту.
55       g_collision_cnt := g_collision_cnt+1;
56     end loop;
57
58   --   Если мы оказались здесь, значит, таблица слишком мала.
59    --   Увеличьте ее.
60     raise_application_error(-20001, 'слишком много хеш-значений в
-> таблице');
61 end;
62
63
64 member function get(p_key in varchar2) return varchar2
65  is
66      l_hash  number :=
67               dbms_utility.get_hash_value(p_key, 1, g_hash_size);
68  begin
```

Если необходимо получить значение, мы обращаемся к элементу индекса, в котором это значение хранится, а в случае возникновения конфликта, просматриваем до 1000

следующих элементов. Поиск прекращается раньше, если обнаружится пустой слот (известно, что нужной записи после этого значения быть не может):

```
71       for i in l_hash .. least(l_hash+1000, nvl(g_hash_table.count,0))
72      loop
73          — Если обнаружили ПУСТОЙ слот, мы ЗНАЕМ, что искомого
74          — значения в таблице нет, поскольку он должен был быть
-> помещен в этот слот.
75         if (g_hash_table(i) is NULL)
76         then
77             return NULL;
78        end if;
79
80         — Если ключ найден, вернуть соответствующее значение.
81         if(g_hash_table(i).key = p_key)
82         then
83             return g_hash_table(i).val;
84        end if;
85      end loop;
86
87      — Ключа в таблице нет. Завершаем работу.
88      return null;
89 end;
90
```

Последняя процедура используется для выдачи полезной информации, например, о количестве выделенных и использованных слотов, а также о количестве конфликтов. Учтите, что количество конфликтов может превосходить размер таблицы!

```
97  member procedure print_stats
98  is
99      l_used number default 0;
100 begin
101     for i in 1 .. nvl(g_hash_table.count,0)
102     loop
103        if (g_hash_table(i) is not null)
104        then
105            l_used := l_used + 1;
106       end if;
107     end loop;
108
109     dbms_output.put_line('Таблица увеличена до....' ||
110                          g_hash_table.count);
111     dbms_output.put_line( 'Мы      используем.....' ||
112                          l_used);
113    dbms_output.put_line('Количество     конфликтов...' ||
114                          g_collision_cnt);
115 end;
116
117 end;
118 /

Type body created.
```

Как видите, мы использовали функцию **GET_HASH_VALUE** для получения по строке числа, которое можно использовать для индексации табличного типа и выборки значения. Теперь можно рассмотреть, как используется этот новый тип:

```
tkyte@TKYTE816> declare
  2    l_hashTblhashTableType :=hashTableType.new(power(2,7));
  3  begin
  4     for x in (select username, created from all_users)
  5      loop
  6          l_hashTbl.put(x.username, x.created);
  7      end loop;
  8
  9     for x in (select username, to_char(created) created,
 10                      l_hashTbl.get(username) hash
 11                 from all_users)
 12      loop
 13        if (nvl(x.created, 'x') onvl(x.hash,'x'))
 14         then
 15            raise program_error;
 16        end if;
 17      end loop;
 18
 19     l_hashTbl.print_stats;
 20 end;
 21  /
Таблица увеличена до. . . . .120
Используется. . . . . .17
Количество конфликтов....1

PL/SQL procedure successfully completed.
```

Вот и все. Мы расширили возможности языка PL/SQL, добавив хеш-таблицу на базе стандартных пакетов.

Резюме

Мы завершаем обзор основных подпрограмм пакета **DBMS_UTILITY.** Многие из них, в частности **GET_TIME, GET_PARAMETER_VALUE, GET_HASH_VALUE** и **FORMAT_CALL_STACK,** входят в мой список "часто даваемых ответов". Это означает, что именно они необходимы для решения многих проблем. Пользователи просто не знают о существовании этих подпрограмм.

Пакет UTL_FILE

Стандартный пакет **UTL_FILE** позволяет читать и создавать текстовые файлы в файловой системе сервера в среде **PL/SQL.** Здесь существенны следующие ключевые слова.

- **Текстовые файлы.** Пакет **UTL_FILE** позволяет читать и создавать простые текстовые файлы. В частности, его нельзя использовать для чтения или создания двоичных файлов. Специальные символы, содержащиеся в двоичных данных, приводят к некорректной работе пакета **UTL_FILE.**
- **В файловой системе сервера.** Пакет **UTL_FILE** позволяет читать и записывать файлы только в файловой системе сервера баз данных. Он не позволяет читать или записывать в файловую систему компьютера, на котором работает клиент, если последний не подключен локально к серверу.

Пакет **UTL_FILE** подходит для создания отчетов и сброса данных из базы в текстовые файлы, а также для чтения и загрузки данных. В главе 9 первой части книги представлен полный пример использования пакета **UTL_FILE** для создания текстового файла в формате, упрощающем загрузку. Пакет **UTL_FILE** также помогает при отладке. В главе 21, посвященной средствам тщательного контроля доступа, представлен пакет **DEBUG.** Этот пакет интенсивно использует средства пакета **UTL_FILE** для записи сообщений в файловую систему.

Пакет **UTL_FILE** — очень полезен, но, используя его, нужно помнить об определенных ограничениях. В противном случае результаты могут оказаться некорректными (причем, выявиться это может не при тестировании, а при внедрении) и вызовут разочарование.

Я рассмотрю ряд проблем, часто встречающихся при использовании пакета **UTL_FILE, в** том числе:

- установку параметра инициализации **UTL_FILE_DIR;**
- доступ к дискам других компьютеров (сетевым дискам) в среде Windows (проблем с этим в среде Unix нет);
- обработку исключительных ситуаций, возникающих при использовании пакета **UTL_FILE;**
- применение пакета **UTL_FILE** для периодического пересоздания статических Web-страниц;
- печально известное ограничение — 1023 байта;
- получение списка файлов каталога, чтобы можно было обработать все находящиеся в нем файлы.

Параметр инициализации UTL_FILE_DIR

Это наиболее существенная особенность использования пакета **UTL_FILE,** который всегда работает от имени пользователя — владельца СУБД Oracle, — ввод-вывод выполняет выделенный или разделяемый сервер, всегда работающий от имени пользователя **oracle.** С учетом того, что работа с файлами выполняется от имени пользователя **oracle,** а пользователь **oracle** может читать и записывать файлы данных, файлы конфигурации и т.п., — не стоит разрешать доступ с помощью средств **UTL_FILE** ко всем каталогам. В файле параметров инициализации необходимость явно перечислять каталоги, к которым необходим доступ на запись, — это средство защиты, а не излишняя сложность. Если бы пакет **UTL_FILE** позволял записывать файлы в доступных пользователю **oracle** каталогах, то любой пользователь с помощью функции **UTL_FILE.FOPEN** смог бы переписать файлы данных системы. Это, конечно же, недопустимо. Поэтому администратор базы данных должен явно открывать доступ к конкретным каталогам. Нельзя разрешить доступ к корневому каталогу поддерева, позволив тем самым обращаться ко всем каталогам этого поддерева, — надо явно перечислить все каталоги, в которых предполагается чтение и изменение файлов с помощью пакета **UTL_FILE.**

Следует помнить, что изменять параметр инициализации в процессе работы сервера нельзя. Для добавления или удаления каталога необходимо перезапускать экземпляр.

Параметр инициализации **UTL_FILE_DIR** можно задавать одним из следующих способов:

```
utl_file_dir = (c:\temp,c:\temp2)
```

или

```
utl_file_dir = c:\temp
utl_file_dir = c:\temp2
```

Другими словами, можно либо задавать список каталогов через запятую, в круглых скобках, либо перечислять каталоги по одному в последовательных строках. Главное здесь — "в последовательных строках". Если файл параметров инициализации **(init.ora)** завершается следующими строками:

```
utl_file_dir = c:\temp
timed_statistics=true
utl_file_dir = c:\temp2
```

учтена будет только последняя запись — **UTL_FILE_DIR**. Первый каталог будет проигнорирован. Это может сбивать с толку, поскольку никаких предупреждающих сообщений или записей в файле alert.log, свидетельствующих об игнорировании записи **UTL_FILE_DIR**, не появляется. Все записи **UTL_FILE_DIR** в файле параметров инициализации должны идти подряд.

Хочу предупредить об одной особенности использования параметра инициализации на платформе Windows. Если добавить к значению параметра **UTL_FILE_DIR** завершающую обратную косую черту (\), например, так:

```
utl_file_dir = c:\temp\
utl_file_dir = c:\temp2
```

при запуске сервера будет выдано следующее сообщение об ошибке:

```
SVRMGR> Startup
LRM-00116: syntax error at 'c:\terrputl_file_' following '='
LRM-00113: error when processing file
-> 'C:\oracle\admin\tkyte816\pfile\init.ora'
ORA-01078: failure in processing system parameters
```

Дело в том, что символ \ является управляющим, если стоит последним в строке файла параметров инициализации. Он позволяет продолжить длинную запись на следующей строке. Чтобы избежать такой конкатенации строк, нужно просто указывать две обратных косых подряд:

```
utl_file_dir = c:\temp\\
utl_file_dir = c:\oracle
```

И последняя особенность этого параметра инициализации. Если в файле параметров инициализации использована завершающая обратная косая черта в имени каталога, необходимо указывать завершающую обратную косую и в вызовах функции fopen. Если обратная косая не указана в файле параметров инициализации, не нужно ее задавать и в вызовах функции fopen. Параметр функции fopen, задающий имя каталога, должен буквально совпадать с одним из значений, заданных в файле параметров инициализации.

Обращение к сетевым дискам в Windows

Эта задача часто вызывает затруднения, особенно у тех, кто привык работать в ОС Unix. В ОС Unix смонтированное устройство (например, смонтированный клиентом NFS удаленный диск) немедленно становится доступным всем пользователям, независимо от того, в каком сеансе оно смонтировано. У каждого пользователя могут быть свои права доступа к нему, но смонтированный диск — это атрибут системы, а не конкретного сеанса.

В среде Windows все не так. На сервере может работать несколько пользовательских сеансов, каждый со своим набором доступных "дисков". Может оказаться, что, зарегистрировавшись, я обнаружу сетевой ресурс — диск D:, физически находящийся на дру-

гой машине. Но это не означает, что процесс, работающий на этой машине, сможет увидеть этот диск. Вот тут и возникают проблемы.

Многие пользователи регистрируются на сервере и видят диск D:. Они включают в файл параметров конфигурации запись **UTL_FILE_dir** = d:\reports — каталог, в котором предполагается создание отчетов с помощью средств пакета **UTL_FILE**. При выполнении, однако, они получают сообщение об ошибке:

```
ERROR at line 1:
ORA-06510: PL/SQL: unhandleduser-definedexception
ORA-06512: at "SYS.UTL_FILE", line 98
ORA-06512: at "SYS.UTL_FILE", line 157
```

С помощью обработчика исключительных ситуаций (ниже представлен код, который я обычно использую) это сообщение можно сделать более информативным, например:

```
ERROR at line 1:
ORA-20001: INVALID_PATH: File location or filename was invalid.
ORA-06512: at "TKYTE.CSV", line 51
ORA-06512: at line 2
```

Итак, с точки зрения пользователя каталог **D:\reports** вполне допустим. Его можно найти с помощью программы **Проводник (Explorer).** Можно открыть окно командной строки DOS и обнаружить в нем этот каталог. Только СУБД Oracle его не видит. Причина в том, что при запуске системы диска D: нет и, более того, учетная запись, от имени которой работает сервер Oracle, по умолчанию вообще не может обращаться к сетевым ресурсам. Пытайтесь сколько угодно, монтируйте диск любым способом — сервер Oracle его не увидит.

При создании экземпляра Oracle службы, поддерживающие его, настраиваются для регистрации в системе от имени учетной записи SYSTEM (как системные), а эта учетная запись имеет ограниченные привилегии, и домены Windows NT ей недоступны. Чтобы обращаться к другой машине, работающей под управлением Windows NT, служба **OracleServiceXXXX** должна зарегистрироваться в соответствующем домене Windows NT от имени пользователя, имеющего доступ к диску, который предполагается использовать с помощью пакета **UTL_FILE**.

Чтобы изменить стандартные параметры регистрации для служб Oracle, выберите (в ОС Windows NT):

Control Panel | Services | OracleServiceXXXX | Startup | Log On As; (где XXXX — имя экземпляра)

В ОС Windows 2000 следует выбрать:

Control Panel | Administrative Tools | Services | OracleServiceXXXX | Properties | Log On Tab; (где XXXX — имя экземпляра)

Выберите переключатель **This Account** и введите соответствующую информацию о регистрации в домене. После настройки служб для работы от имени пользователя с соответствующими привилегиями, параметр **UTL_FILE_DIR** можно задать одним из двух способов.

- **С помощью сопоставленного диска.** Чтобы использовать сопоставленный удаленному ресурсу диск, необходимо, чтобы пользователь, от имени которого запуска-

ется служба, настроил диск, указанный в значении параметра **UTL_FILE_DIR, и** был зарегистрирован на сервере при использовании пакета **UTL_FILE.**

- **С помощью универсальных соглашений по именованию.** Использование универсальных соглашений по именованию (Universal Naming Conventions — UNC) предпочтительнее сопоставления дисков, поскольку не требует регистрации пользователя. При этом значение параметра инициализации **UTL_FILE_DIR** задается в виде **\\<имя машины >\<имя общего ресурса >\<путь>>.**

Естественно, сервер Oracle после изменения свойств сетевой службы необходимо перезапустить.

Обработка исключительных ситуаций

При возникновении ошибки пакет **UTL_FILE** возбуждает исключительную ситуацию. К сожалению, он возбуждает исключительные ситуации, задаваемые пользователем, — они определены в спецификации пакета. Если эти исключительные ситуации не перехвачены по имени, выдаются совершенно бесполезные сообщения об ошибках:

```
ERROR at line 1:
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at "SYS.UTL_FILE", line 98
ORA-06512: at "SYS.UTL_FILE", line 157
```

О самой ошибке в этом сообщении ничего не сказано. Для решения этой проблемы надо включить обращения к пакету **UTL_FILE** в блок обработки исключительных ситуаций, где каждая из них перехватывается по имени. Я предпочитаю преобразовывать исключительные ситуации в вызовы процедуры **RAISE_APPLICATION_ERROR**. Это позволяет задать код ошибки **ORA-** и выдать более информативное сообщение об ошибке. В предыдущем примере мы использовали этот прием для преобразования представленного выше сообщения об ошибке в следующее:

```
ORA-20001: INVALID_PATH: File location or filename was invalid.
```

Это сообщение намного полезнее. Я всегда использую блок обработки исключительных ситуаций следующего вида:*

```
exception
    when utl_file.invalid_path then
        raise_application_error(-20001, — ошибка в имени файла
       'INVALID_PATH: File location or filename was invalid.');
    when utl_file.invalid_mode then
        raise_application_error(-20002, — недопустимое значение режима
-> открытия файла
      'INVALID_MODE: The open_mode parameter in FOPEN was
       invalid.');
    when utl_file.invalid_filehandle then
        raise_application_error(-20002, — недопустимый дескриптор файла
         'INVALID_FILEHANDLE: The file handle was invalid.');
```

** Исходные сообщения об ошибках приводятся на английском, а в строке с кодом ошибки добавлен комментарий на русском языке с описанием причины ее возникновения.* **- Прим. научн. ред.**

```
    when utl_file.invalid_operation then
        raise_application_error(-20003, — файл нельзя открыть или
-> обработать в указанном режиме
        'INVALID_OPERATION: The file could not be opened or
        operated on as requested.');
    when utl_file.read_error then
        raise_application_error(-20004, — в ходе чтения произошла ошибка
-> операционной системы
        'READ_ERROR: An operating system error occurred during
        the read operation.');
    when utl_file.write_error then
        raise_application_error(-20005, — в ходе записи произошла ошибка
-> операционной системы
            'WRITE_ERROR: An operating system error occurred
             during the write operation.');
    when utl_file.internal_error then
        raise_application_error(-20006, — неизвестная ошибка в PL/SQL
            'INTERNAL_ERROR: An unspecified error in PL/SQL.');
end;
```

Я записал этот блок в отдельный небольшой файл и добавляю в каждую подпрограмму, использующую пакет UTL_FILE, чтобы автоматически перехватывать и "переименовывать" исключительные ситуации.

Как сбросить Web-страницу на диск?

Этот вопрос задают так часто, что я решил дать ответ здесь. Предполагается, что используется Oracle WebDB, Oracle Portal или другие процедуры на базе средств Web Toolkit (пакетов htp). Хотелось бы не генерировать динамически отчет, который можно получить с помощью этих средств, отдельно для каждого пользователя, а периодически, один раз в X часов или минут, создавать статический файл с отчетом. Именно так я и генерирую начальную страницу на своем сайте. Вместо того чтобы генерировать ее динамически для каждого из тысяч поступающих за этот период обращений, раз в 5 минут я генерирую статическую начальную страницу заново на основе динамических данных. Это существенно сокращает ресурсы, необходимые для поддержки сайта. Такой прием я применяю для популярных динамических станиц, базовые данные которых меняются сравнительно редко.

Ниже представлена универсальная процедура, которую я для этого использую:

```
create or replace procedure dump_page(p_dir     in varchar2,
                                       p_fname   in varchar2)
is
    l_thePage       htp.htbuf_arr;
    l_output        utl_file.file_type;
    l_lines         number default 99999999;
begin
    l_output := utl_file.fopen(p_dir, p_fname, 'w', 32000);

    owa.get_page(l_thePage, l_lines);

    for i in 1 .. l_lines loop
```

```
        utl_file.put(l_output, l_thePage(i));
    end loop;

    utl_file.fclose(l_output);
end dump_page;
/
```

Все очень просто. Необходимо открыть файл, получить HTML-страницу, выдать каждую ее строку в файл и закрыть его. Если вызвать эту процедуру после WebDB-процедуры, она сохранит результат работы WebDB-процедуры в заданном файле.

Единственная проблема при использовании этого подхода состоит в том, что WebDB-процедура выполняется непосредственно, а не через Web. Если в коде WebDB-процедуры используется среда CGI, эта процедура не выполнится, поскольку среда не настроена. Для решения этой проблемы достаточно использовать небольшой фрагмент кода для настройки среды:

```
declare
    nm  owa.vc_arr;
    vl  owa.vc_arr;
begin
    nm(1) := 'SERVER_PORT';
    vl(1) := '80';
    owa.init_cgi_env(nm.count, nm, vl);
    — здесь выполните необходимую webdb-процедуру
    dump_page('directory', 'filename');
end;
/
```

Например, если WebDB-процедура проверяет, запущена ли она с порта 80 на сервере, надо создать для нее соответствующую среду. В этом блоке надо задать и все остальные переменные среды, существенные для приложения.

Теперь осталось только перечитать раздел, посвященный пакету **DBMS_JOB,** и обеспечить периодическое выполнение этого блока кода.

Ограничение длины строки — 1023 байт

Когда-то у пакета **UTL_FILE** было ограничение длины записываемой в файл строки: не более 1023 байт. В противном случае возбуждалась исключительная ситуация, и процедура пакета **UTL_FILE** не выполнялась. К счастью, в версии Oracle 8.0.5 добавлена новая версия функции **FOPEN,** позволяющая при открытии файла задавать максимальнуюдлину строки вплоть до 32 Кбайт. 32 Кбайт — максимальный размер переменной в PL/SQL, и такой длины в большинстве случаев хватает.

К сожалению, в документации эта перегруженная версия функции **FOPEN** описана через несколько страниц **после** исходной функции. Поэтому многие пользователи об этой возможности и не подозревают. Я по-прежнему получаю много вопросов об этом, хотя сейчас используются версии, начиная с 8.1.7. Пользователя не замечают перегруженную версию функции **FOPEN;** они сталкиваются с ограничением и ищут способы обойти его. Ответ простой, но, чтобы найти его, надо прочитать описание всех функций пакета **UTL_FILE!**

Для решения этой проблемы необходимо использовать пакет **UTL_FILE** так, как это делалось в представленной выше процедуре **DUMP_PAGE**. Четвертый параметр вызова функции **UTL_FILE.FOPEN** задает максимальную длину строки текста, которую предполагается использовать. Я допускал использование строк длиной до 32000 байт.

Чтение содержимого каталога

Этой возможности в пакете **UTL_FILE** не хватает. Часто необходимо создать периодически выполняющееся задание для просмотра каталога в поисках новых файлов и обработки этих файлов, например загрузки их содержимого в базу данных. К сожалению, стандартного способа прочитать список файлов в каталоге в языке PL/SQL нет. Можно, однако, реализовать эту возможность с помощью небольшого фрагмента кода на языке Java. Следующий пример демонстрирует, как это сделать.

Сначала я создам пользователя с минимальным набором привилегий, необходимых для выполнения действий по загрузке данных и получения списка файлов в каталоге **/tmp**. Если необходимо читать содержимое других каталогов, придется добавить соответствующие вызовы **dbmsJava.grant_permission** (подробнее о них см. в главе 19) или заменить **/tmp** на *, что позволит получить список файлов любого каталога.

```
SQL> connect system/manager

system@DEV816>drop user dirlist cascade;
User dropped.

system@DEV816>grant create session, create table, create procedure
  2  to dirlist identified by dirlist;
Grant succeeded.

system@DEV816> begin
  2     dbms_java.grant_permission
  3       ('DIRLIST',
  4       'java.io.FilePermission',
  5            '/tmp',
  6            'read');
  7  end;
  8  /
PL/SQL procedure successfully completed.
```

Затем, подключившись от имени пользователя **DirList**, мы создаем глобальную временную таблицу в его схеме (для хранения списка файлов каталога). Так, через временную таблицу, мы сможем получить результаты выполнения хранимой процедуры на языке Java в вызывающей среде. Можно использовать для этого и другие способы (строки, массивы и т.п.).

```
SQL> connect dirlist/dirlist
Connected.

dirlist@DEV816>create global temporary table DIR_LIST
  2  (filename varchar2(255))
  3  on commit delete rows
  4  /
Table created.
```

Теперь создадим хранимую процедуру на языке Java для получения списка файлов в указанном каталоге. Чтобы упростить программирование, я использую средства препроцессора SQLJ — это позволяет избежать написания большого количества вызовов JDBC:

```
dirlist@DEV816> create or replace
  2    and compile java source named "DirList"
  3  as
  4  import java.io.*;
  5  import java.sql.*;
  6
  7  public class DirList
  8  {
  9  public static void getList(String directory)
 10                    throws SQLException
 11  {
 12      File path = new File(directory);
 13     String[] list = path.list();
 14      String element;
 15
 16     for (int i = 0; i < list.length; i++)
 17      {
 18         element = list[i];
 19         #sql { INSERT INTO DIR_LIST (FILENAME)
 20                VALUES (:element) };
 21      }
 22  }
 23
 24  }
 25  /
Java created.
```

Затем необходимо создать процедуру сопоставления, связывающую языки PL/SQL и Java. Она достаточно проста:

```
dirlisteDEV816> create or replace
  2  procedure get_dir_list(p_directory in varchar2)
  3  as language java
  4  name 'DirList.getList(java.lang.String)';
  5  /
Procedure created.
```

Теперь можно использовать процедуру get_dir_list:

```
dirlist@DEV816> exec get_dir_list('\tmp');
PL/SQL procedure successfully completed.
dirlist@DEV816> select * from dir_list where rownum < 5;

FILENAME
------------------
lost+found
.rpc_door
ps_data
.pcmcia
```

Вот и все. В соответствующей временной таблице теперь можно получить список файлов каталога. К данным таблицы можно применять фильтры, например **LIKE**, или сортировать результаты.

Резюме

Пакет **UTL_FILE** — замечательная утилита, которая пригодится во многих приложениях. В этом разделе мы рассмотрели как настроить сервер для использования средств пакета **UTL_FILE**, и описали особенности его работы. Мы рассмотрели наиболее часто возникающие проблемы при использовании пакета **UTL_FILE**, в частности обращение к сетевым дискам в среде Windows, ограничение длины строки 1023 байтами, и обработку исключительных ситуаций. Для каждой из этих проблем были представлены решения. Мы также изучили ряд утилит, которые можно создать с помощью пакета **UTL_FILE**, в частности процедуру **UNLOADER**, описанную в главе 9, средства чтения списка файлов каталога и сброса Web-страницы на диск.

Пакет UTL_HTTP

В этом разделе мы рассмотрим, когда и как использовать пакет **UTL_HTTP.** Кроме того, я хочу представить новую расширенную версию пакета **UTL_HTTP,** созданную на основе типа **SocketType,** который рассматривается в разделе, посвященном пакету **UTL_TCP.** Его производительность сравнима с обеспечиваемой стандартным пакетом **UTL_HTTP,** а возможности — намного шире.

Стандартный пакет **UTL_HTTP,** поставляемый вместе с сервером, реализует весьма упрощенный подход. Он содержит две **функции.**

- **UTL_HTTP.REQUEST:** возвращает до 2000 первых байт содержимого с заданным адресом URL.
- **UTL_HTTP.REQUEST_PIECES:** возвращает PL/SQL-таблицу элементов типа **VARCHAR2(2000).** Если конкатенировать последовательно все элементы, будет получено содержимое соответствующей страницы.

В пакете **UTL_HTTP,** однако, не хватает многих возможностей.

- Нельзя проверить заголовки **HTTP.** Это не позволяет выдавать сообщения об ошибках. Нельзя, например, различить ошибки доступа **Not Found** и **Unauthorized.**
- Нельзя пересылать информацию на Web-сервер с помощью метода **POST.** Можно использовать только метод GET. Кроме того, не поддерживается метод **HEAD** протокола HTTP.
- С помощью пакета **UTL_HTTP** нельзя получать двоичные данные.

- Интерфейс запроса страницы по частям **(REQUEST_PIECES)** неочевиден — намного проще было бы использовать данные типа **CLOB** или **BLOB** для возврата данных в виде "потока" (что обеспечило бы заодно и доступ к двоичным данным).
- Пакет не поддерживает "ключики" (cookies).
- Пакет не поддерживает даже простейшую аутентификацию.
- В пакете нет методов кодирования адреса URL.

Одна из возможностей, которые пакет **UTL_HTTP** поддерживает, — это использование протокола SSL. С помощью диспетчера Oracle Wallet можно выполнять запросы по протоколу HTTPS (HTTPS — это реализация протокола HTTP поверх SSL). Я продемонстрирую использование пакета **UTL_HTTP** для доступа по протоколу SSL, но соответствующие возможности в пакете **HTTP_PKG** реализовывать не будем. Полный текст пакета **HTTP_PKG** вследствие большого размера в этом разделе не приводится; он доступен на сайте издательства Wrox по адресу **http://www.wrox.com.**

Возможности пакета UTL_HTTP

Рассмотрим сначала функциональные возможности пакета **UTL_HTTP,** поскольку предполагается реализовать аналогичные в пакете **HTTP_PKG.** Простейший вариант использования средств пакета **UTL_HTTP** представлен далее. В этом примере **myserver** — имя моего Web-сервера. Вы, разумеется, должны выполнить этот пример, обращаясь в Web-серверу, который вам доступен:

```
ops$tkyte@DEV816>selectutl_http.request('http://myserver/')fromdual;
UTL_HTTP.REQUEST('HTTP://MYSERVER/')

<HTML>
<HEAD>
<TITLE>OracleServiceIndustries</TITLE>
</HEAD>
<FRAMESET COLS="130,*" border=0>
<FRAME SRC="navtest.html" NAME="sidebar" frameborder=0>
<FRAME SRC="folder_home.html" NAME="body" frameborder="0"
marginheight="0" marginwidth="0">
</FRAMESET>
</BODY>
</HTML>
```

Можно просто вызвать функцию **UTL_HTTP.REQUEST**, передав ей адрес URL. Пакет **UTL_HTTP** подключится к соответствующему Web-серверу и запросит **(GET)** указанную страницу, а затем вернет первые 2000 ее символов. Как уже было сказано, не пытайтесь использовать указанный в примере адрес URL — это адрес моего рабочего Web-сервера в корпорации Oracle. Вы не сможете к нему добраться — по истечении времени ожидания будет выдано сообщение об ошибке.

Большинство сетей сегодня защищено брандмауэрами (межсетевыми экранами). Если необходимая страница доступна только через промежуточный сервер брандмауэра, я мог

бы запросить ее и так. Обсуждение брандмауэров и промежуточных серверов выходит за рамки этой книги. Однако если известно имя хоста, на котором работает промежуточный сервер, можно выбрать через него страницу из Internet следующим образом:

```
ops$tkyte@DEV816> select utl_http.request('http://www.yahoo.com', 'www-
-> proxy') from dual;

UTL_HTTP.REQUEST('HTTP://WWW.YAHOO.COM','WWW-PROXY')

<html><head><title>Yahoo!</title><base href=http://www.yahoo.com/><meta
http-equiv="PICS-Label" content='(PICS-1.1 http://www.rsac.org/
ratingsv01.html" 1 gen true for "http://www.yahoo.com" r (n 0 s 0 v 0 1
0))'></head><bodyXcenter><form action=http://search.yahoo.com/bin/
search><map name=m><area coords="0,0,52,52" href=r/a1><area
coords="53,0,121,52" href=r/p1><area coords="122,0,191,52" href=r
```

Второй параметр функций UTL_HTTP.REQUEST и REQUEST_PIECES — имя промежуточного сервера. Если промежуточный сервер работает не на стандартном порту (80), можно добавить номер порта следующим образом (в коде промежуточный сервер — myserver, и работает он на порту 8000):

```
ops$tkyte@DEV816> select utl_http.request('http://www.yahoo.com',
  2  'myserver:8000') from dual
  3  /

UTL_HTTP.REQUEST('HTTP://WWW.YAHOO.COM','MYSERVER:8000')

<html><head><title>Yahoo!</title><base href=http://www.yahoo.com/
```

Итак, добавив :8000 к имени промежуточного сервера, мы смогли к нему подключиться. Теперь давайте рассмотрим использование функции REQUEST_PIECES:

```
ops$tkyte@DEV816> declare
  2      pieces  utl_http.html_pieces;
  3      n       number default 0;
  4      l_start number default dbms_utility.get_time;
  5  begin
  6      pieces :=
  7           utl_http.request_pieces(url => 'http://vww.oracle.com/',
  8                                   max_pieces => 99999,
  9                                   proxy=> 'www-proxy');
 10     for i in 1 .. pieces.count
 11     loop
 12         loop
 13            exit when pieces(i) is null;
 14            dbms_output.put_line(substr(pieces(i),1,255));
 15           pieces(i) := substr(pieces(i), 256);
 16         end loop;
 17     end loop;
 18 end;
 19 /

<head>
```

```
<title>Oracle Corporation</title>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1">
<meta name="description" content="Oracle Corporation provides the
software that
powers the Internet. For more information about Oracle, pleas
e call 650/506-7000.">
```

Функцию **REQUEST_PIECES** нельзя вызывать в SQL-операторах, поскольку она возвращает не SQL-тип, а PL/SQL-таблицу. Поэтому функцию **REQUEST_PIECES** можно использовать только в PL/SQL-блоках. В представленном выше примере мы запросили первые 99999 фрагментов Web-страницы **http://www.oracle.com/,** каждый из которых длиной 2000 байт. При этом используется промежуточный сервер **www-proxy.** Мы должны передать функции **REQUEST_PIECES** количество фрагментов по 2000 байт, которые мы хотим получить (обычно я задаю очень большое значение, поскольку хочу получить всю страницу). Если на запрошенной странице вас всегда интересуют первые 5000 байт информации, для ее получения можно запросить 3 фрагмента.

Добавление поддержки протокола SSL в пакете UTL_HTTP

Пакет **UTL_HTTP** также поддерживает использование протокола SSL (Secure Sockets Layer — уровень защищенных сокетов). Если вы не знакомы с протоколом SSL и его назначением, прочитайте краткое описание на странице **http://www.rsasecurity.com/rsalabs/faq/5-1-2.html.** Обе функции, **REQUEST** и **REQUEST_PIECES,** пакета **UTL_HTTP** поддерживают получение URL, защищенных протоколом SSL. Однако в документации это описано, мягко говоря, в общих чертах. Поддержка протокола SSL обеспечивается путем передачи двух дополнительных параметров функциям **UTL_HTTP.REQUEST** и **UTL_HTTP.REQUEST_PIECES.** Это параметры **WALLET_PATH** и **WALLET_PASSWORD.**

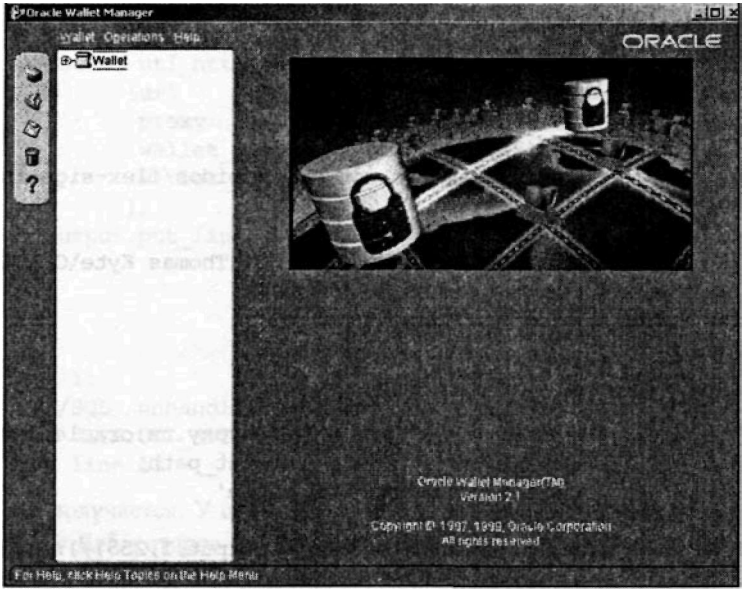
Сервер Oracle использует понятие "бумажник" (wallet) по аналогии с тем, как люди хранят важные документы, — паспорт, водительские права и кредитные карточки принято держать в бумажнике на случай, если их придется предъявлять. Сервер Oracle держит в своем "бумажнике" регистрационную информацию для протокола SSL. Параметр **WALLET_PATH** задает каталог на машине, где работает сервер базы данных, в котором хранится этот "бумажник". Доступ к бумажнику защищен паролем, чтобы другой пользователь не мог зарегистрироваться от вашего имени. Пароль передается как параметр **WALLET_PASSWORD** — он используется для получения доступа к "бумажнику". Пароль предотвращает копирование каталога "бумажника" другими пользователями, пытающимися выдать себя за вас, поскольку без него они не смогут работать с этим "бумажником". Это аналогично использованию PIN-кода при снятии денег с карточки в банкомате. Если кто-то украдет вашу кредитку, чтобы добраться до счетов ему понадобится ввести ваш PIN-код.

Метафора "бумажника" используется не только сервером Oracle, но и Web-браузерами. Интересно, как при подключении к сайту, например, **http://www.amazon.com,** вы

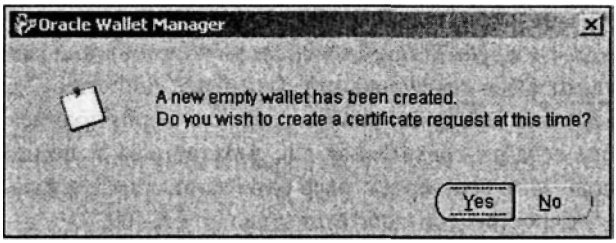
можете удостовериться, что это действительно служба **Amazon.com?** Необходимо получить их сертификат, с цифровой подписью авторитетной организации. Такие организации иногда называют бюро сертификации (Certificate Authority — CA). Откуда браузер или сервер базы данных знает, что можно доверять бюро, подписавшему сертификат? Например, я могу создать сертификат для **Amazon.com** и подписать его от имени **hackerAttackers.com.** Мой браузер и сервер базы данных не должны принять такой сертификат, хотя он полностью соответствует стандарту X.509.

Ответ на вопрос о доверии связан с использованием "бумажника", в котором хранится набор надежных сертификатов. Надежным считается сертификат, выданный бюро сертификации, которому вы доверяете. В "бумажник" Oracle входит ряд сертификатов, которым обычно доверяют. Вы можете добавлять сертификаты при необходимости. То же самое делает и браузер. При подключении к сайту, сертификата для которого нет в "бумажнике" браузера, появляется окно, уведомляющее об этом, и мастер подключения, позволяющий продолжить подключение или отказаться от него.

Давайте рассмотрим несколько примеров использования протокола SSL. Сначала надо создать новый "бумажник". Для этого надо вызвать программу OWM (Oracle Wallet Manager) в ОС UNIX или выбрать ее из меню кнопки Пуск в Windows (она находится в группе **ORACLE HOME|NETWORK ADMINISTRATION).** При этом появится окно следующего вида:



Достаточно щелкнуть на пиктограмме нового "бумажника" (зеленый куб), которая находится слева. Будет выдано приглашение на ввод пароля для "бумажника" — задается новый пароль, так что можете ввести любой. Может быть выдано предупреждение о том, что каталог не существует. Проигнорируйте его. Так и должно быть, если раньше "бумажники" не создавались. Затем программа OWM предложит создать новый запрос сертификата:



Его создавать не надо. Запрос сертификата означает, что вы хотите получить сертификат для своего компьютера. Он будет использоваться протоколом SSL v.3, когда серверу понадобится идентифицировать клиента. Большинство Web-сайтов аутентифицирует пользователей не по сертификатам, а по регистрационному имени и паролю. Причина в том, что сайту электронной коммерции все равно, кто покупает товар, — лишь бы деньги заплатили. А вот вам важно точно знать, кому посылаются деньги (и информация о кредитной карточке), поэтому мы используем протокол SSL v.2 для идентификации сервера, например, **Amazon.com** и для шифрования данных. Поэтому щелкните на кнопке NO в ответ на это предложение, сохраните "бумажник", щелкнув на пиктограмме SAVE WALLET (желтая дискета), и продолжайте работу.

Давайте сначала обратимся к сайту **Amazon.com.** Сертификат Amazon был подписан бюро Secure Server Certificate Authority, RSA Data Security, Inc. Это одно из стандартных бюро в "бумажнике" Oracle.

```
tkyte@TKYTE816> declare
  2    l_output long;
  3
  4   l_url varchar2(255). default
  5        'https://www.amazon.com/exec/obidos/flex-sign-in/';
  6
  7   l_wallet_path varchar2(255) default
  8       'file:C:\Documents and Settings\Thomas Kyte\ORACLE\WALLETS';
  9
 10
 11 begin
 12   l_output := utl_http.request
 13               (url            => l_url,
 14               proxy          => 'www-proxy.us.oracle.com',
 15               wallet_path    => l_wallet_path,
 16               wallet_password => 'oracle'
 17              ) ;
 18 dbms_output.put_line(trim(substr(l_output,1,255)));
 19 end;
 20 /


<html>
<head>
<title>Amazon.comErrorPage</title>
</head>
```

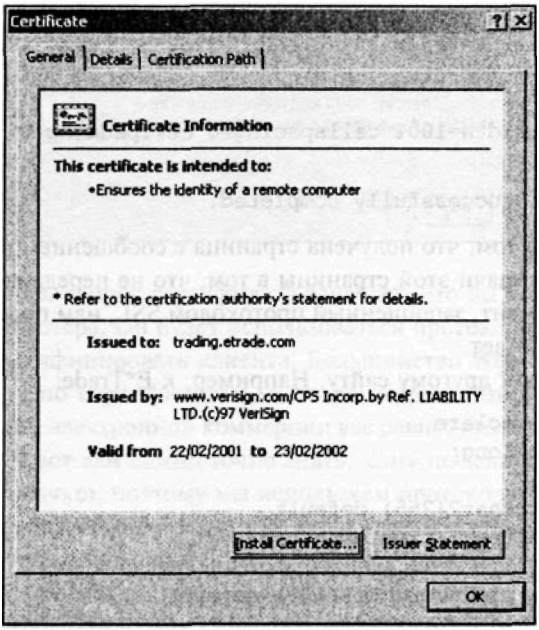
```
<body bgcolor="#FFFFFF"
link="#003399" alink="#FF9933" vlink="#996633" text="#000000"">
<a name="top"><!--Top of
Page--></a>
<table border=0 width=100% cellspacing=0 cellpadding=0>
<tr

PL/SQL procedure successfully completed.
```

Не беспокойтесь о том, что получена страница с сообщением об ошибке; так и должно быть. Причина выдачи этой страницы в том, что не передана информация о сеансе. Мы выбрали документ, защищенный протоколом SSL, нам просто нужно проверить, что подключение работает.

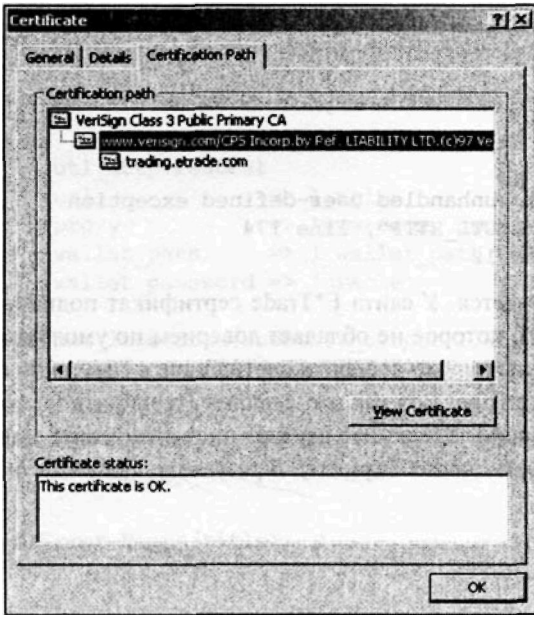
Давайте обратимся к другому сайту. Например, к E*Trade:

```
tkyte@TKYTE816> declare
  2     l_output  long;
  3
  4     l_url varchar2(255) default
  5           'https://trading.etrade.com/';
  6
  7     l_wallet_path varchar2(255) default
  8          'file:C:\Documents and Settings\Thomas Kyte\ORACLE\WALLETS';
  9
 10
 11  begin
 12    l_output  := utl_http.request
 13                (url             => l_url,
 14                 proxy           => 'www-proxy.us.oracle.com',
 15                 wallet_path     => l_wallet_path,
 16                 wallet_password => 'oracle'
 17                );
 18    dbms_output.put_line(trim(substr(l_output,1,255)));
 19  end;
 20  /
declare
*
ERROR at line 1:
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at "SYS.UTL_HTTP", line 174
ORA-06512: at line 12
```

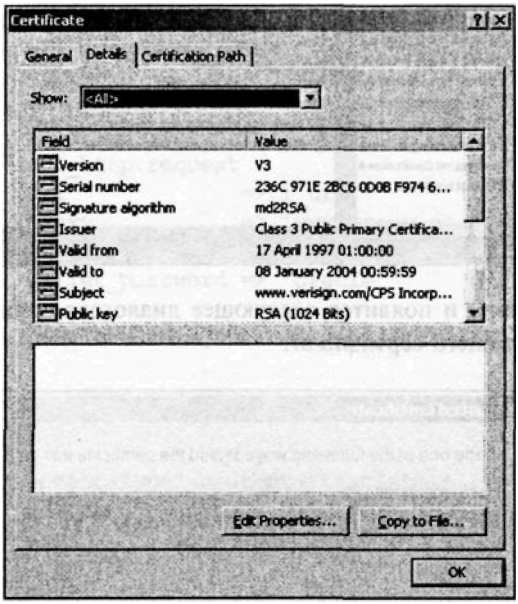
Очевидно, не получается. У сайта E*Trade сертификат подписан бюро **www.verisign com/CPS Incorp.by Ref,** которое не обладает доверием по умолчанию. Чтобы обратиться к этой странице, необходимо добавить сертификат в "бумажник" Oracle — если, конечно, вы доверяете Verisign! Вот как это сделать. Перейдите на соответствующий сайт **(https://trading.etrade.com).** Щелкните дважды на пиктограмме замка в правом нижнем углу (в браузере Microsoft Internet Explorer). В результате появится окно следующего вида:



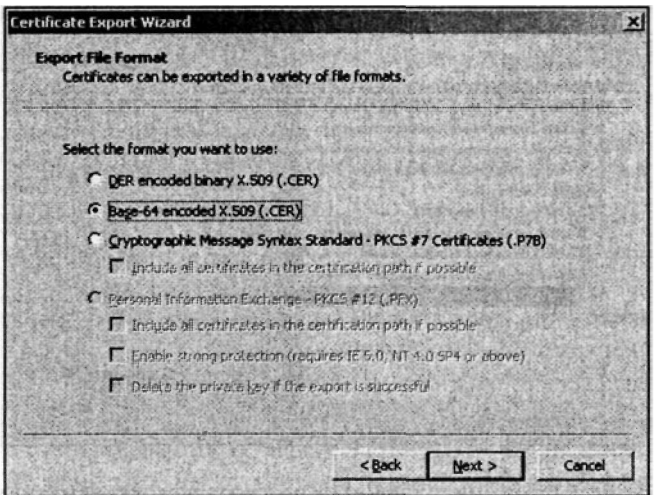
Выберите вкладку **Certification Path** в верхней его части. Здесь представлен соответствующий сертификат E*Trade **(trading.etrade.com),** а также информация о том, кто выдал этот сертификат. Необходимо добавить бюро, подписавшее (выдавшее) сертификат, к списку бюро, сертификатам которых вы доверяете, в "бумажник" Oracle. Сертификат выдало бюро **www.verisign.com/CPS Incorp.by Ref. LIABILITY LTD,** как видно из следующей иерархии:



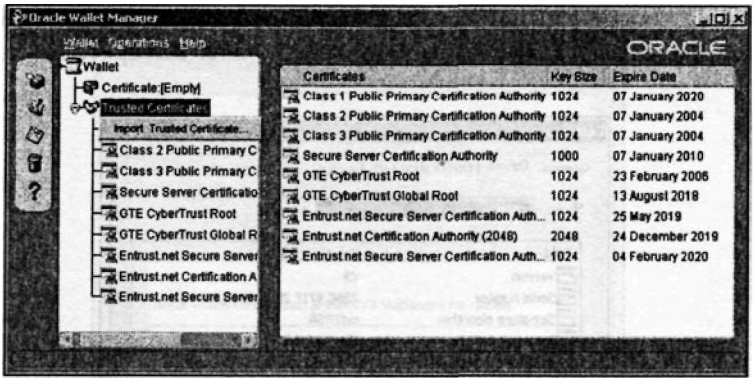
Щелкните на кнопке **View Certificate** при выбранной строке **www.verisign.com/CPS Incorp. by Ref.** Будет выдана информация о сертификате. Выберите вкладку **Details,** и увидите:



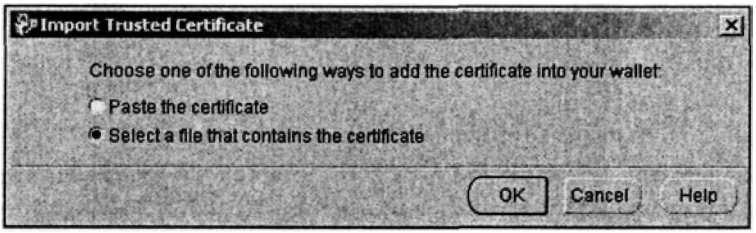
Теперь необходимо щелкнуть на кнопке **Copy to File.** Сохраните файл локально как файл сертификата Base-64 encoded X.509 (CER). Следующий снимок экрана показывает, какой выбор надо сделать; назовите файл как угодно и сохраните его в любом каталоге. Скоро мы его будем импортировать, так что запомните, где вы его сохранили:



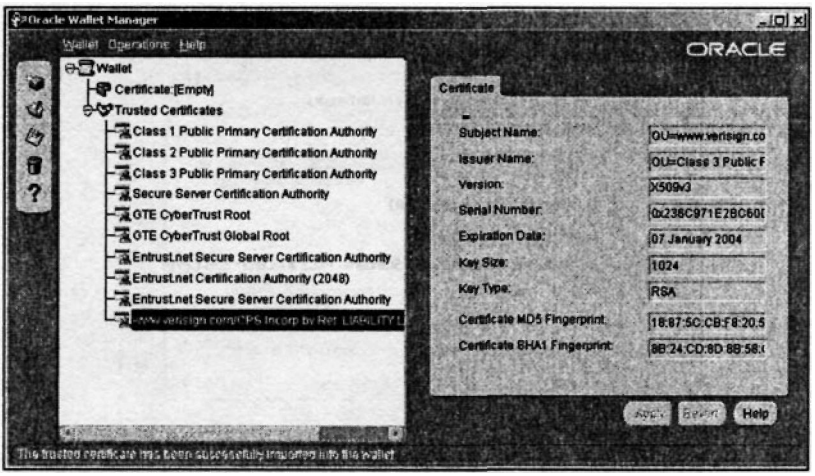
Теперь можно импортировать этот файл в "бумажник" Oracle Wallet. Откройте "бумажник" в программе OWM и щелкните правой кнопкой мыши на **Trusted Certificates;** при этом отобразится меню, в котором есть пункт **import** Trusted Certificate:



Выберите эту опцию, и появится следующее диалоговое окно; нажмите Select для выбора файла, содержащего сертификат.



Используйте появляющееся стандартное диалоговое окно выбора файла для выбора только что сохраненного сертификата. В результате на экране вы должны получить примерно следующее:



Теперь сохраните бумажник, щелкнув на соответствующей пиктограмме (желтая дискета) и попробуйте повторно выполнить пример:

```
tkyte@TKYTE816> declare
  2     l_output long;
```

```
 3
 4    l_url varchar2(255) default
 5        'https://trading.etrade.com/cgi-bin/gx.cgi/AppLogic%2bHome';
 6
 7    l_wallet_path varchar2(255) default
 8        'file:C:\Documents and Settings\Thomas Kyte\ORACLE\WALLETS';
 9
10
11  begin
12    l_output := utl_http.request
13                (url            => l_url,
14                proxy          => 'www-proxy.us.oracle.com',
15                wallet_path    => l_wallet_path,
16               wallet_password => 'oracle'
17               );
18  dbms_output.put_line(trim(substr(l_output,1,255)));
19   end;
20   /
<HTML>
<HEAD>
<META http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<TITLE>E*TRADE</TITLE>
<SCRIPT LANGUAGE="Javascript"
TYPE="text/javascript">
        <!--

        function mac_comment(){
                var
agt=navigator.userAgent.toLowerCase();
                var is_mac

PL/SQL procedure successfully completed.
```

На этот раз все получилось. Теперь вы знаете, как использовать и формировать "бумажник" Oracle для работы по безопасному протоколу **HTTPS.**

Реальное использование пакета UTL_HTTP

Итак, что еще можно сделать с помощью пакета **UTL_HTTP,** кроме получения содержимого Web-страницы? Обычно его также используют, чтобы добиться возможности выполнения внешней программы из PL/SQL — как своего рода команды **HOST.** Поскольку практически любой Web-сервер может выполнять cgi-программы, а пакету **UTL_HTTP** можно передавать адреса URL, можно обеспечить выполнение команд операционной системы из программ на языке PL/SQL, сконфигурировав соответствующие команды для выполнения как cgi-программы на Web-сервере.

Я хочу настроить Web-сервер на IP-адресе **127.0.0.1** — интерфейсе закольцовывания **TCP/IP.** Этот IP-адрес доступен только после физической регистрации на машине, где работает Web-сервер. Таким образом, я смогу создать cgi-программы для моих PL/SQL-программ, которые никто другой выполнить не сможет, если только не взломает сервер — тогда у меня все равно будут серьезные проблемы.

Раньше я использовал этот метод для отправки сообщений электронной почты из PL/SQL-процедур. Предположим, имеется сервер Oracle 8i без поддержки языка Java. Значит, нельзя использовать пакеты **UTL_TCP** и **UTL_SMTP** — оба они предполагают поддержку Java в базе данных. Итак, если нельзя использовать пакеты **UTL_SMTP и/**или **UTL_TCP,** как послать сообщение по электронной почте? С помощью пакетов **UTL_HTTP** и **UTL_FILE** можно создать cgi-программу, которая будет получать один параметр через переменную среды **QUERY_STRING.** Этот параметр будет представлять собой имя файла. Далее мы используем программу **/usr/lib/sendmail** для посылки этого файла (в среде Windows можно использовать для отправки электронной почты общедоступную утилиту **blat,** которую можно найти по адресу **http://www.interiog.com/~tcharron/blat.html).** После необходимых настроек можно будет выполнить хост-команду с помощью вызова:

```
results := utl_http.request(
             'http://127.0.0.1/cgi-bin/smail?filename.text');
             . . .
```

Созданная cgi-программа **smail** будет возвращать Web-страницу, показывающую результат посылки. Я проанализирую содержимое переменной **results**, чтобы узнать, успешно ли отправлено сообщение. Полная реализация всего этого в ОС Unix может иметь вид:

```
scott@ORA8I.WOKLD>create sequence sm_seq
  2  /
Sequencecreated.

scott@ORA8I.WORLD>create or replace procedure sm(p_to in varchar2,
  2                                 p_from in varchar2,
  3                                 p_subject in varchar2,
  4                                 p_body in varchar2)
  5  is
  6      l_output       utl_file.file_type;
  7      l_filename     varchar2(255);
  8      l_request      varchar2(2000);
  9  begin
 10     select 'm' || sm_seq.nextval || '.EMAIL.' || p_to
 11        into l_filename
 12        from dual;
 13
 14     l_output := utl_file.fopen
 15         ('/tmp',         l_filename,  'w', 32000);
 16
 17     utl_file.put_line(l_output, 'From: ' || p_from);
 18     utl_file.put_line(l_output, 'Subject: ' || p_subject);
 19      utl_file.new_line(l_output);
 20      utl_file.put_line(l_output, p_body);
 21      utl_file.new_line(l_output);
 22  utl_file.put_line(l_output,' . ' ) ;
 23
 24      utl_file.fclose(l_output);
```

```
25
26     l_request := utl_http.request
27                  ("http://127.0.0.1/cgi-bin/smail?' || l_filename);
28
29      dbms_output.put_line(l_request);
30  end sm;
31  /

Procedure created.
```

Чтобы понять, почему сообщение имеет именно такой формат, с заголовками From: и To:, обратитесь к разделу приложения, посвященному пакету UTL_SMTP. В этой процедуре для генерации уникального имени файла используется последовательность. Адресата мы включаем в имя файла. Затем записываем сообщение в файл операционной системы. Наконец, используем функцию пакета UTL_HTTP для выполнения соответствующей команды, и передаем ей имя созданного файла. В этом тестовом варианте мы просто выдаем возвращаемый функцией результат; на самом деле следовало бы проверять значение l_result, чтобы определить, успешно отправлено ли сообщение.

Соответствующая cgi-программа может иметь следующий вид:

```
#!/bin/sh

echo "Content-type: text/plain"
echo ""

echo $QUERY_STRING
to=`echo $QUERY_STRING|sed 's/.*EMAIL\.//'`
echo $to

(/usr/lib/sendmail $to < /tmp/$QUERY_STRING) 1 >> /tmp/$$.log 2>&1
cat /tmp/$$.log
rm /tmp/$$.log
rm /tmp/$QUERY_STRING
```

Представленный сценарий командного интерпретатора начинает работу с выдачи заголовков HTTP, которые необходимы для возврата документа, — это и делают первые две команды **echo.** Затем выдается значение переменной QUERY_STRING (именно в нее Web-сервер поместит переданные данные, указанные после ? в адресе URL). Затем с помощью программы **sed** (Stream Editor — потоковый редактор) из строки **QUERY_STRING** выбирается адрес электронной почты, и из имени файла удаляется все до строки EMAIL/ включительно. После этого мы выполняем команду **sendmail** в порожденном командном интерпретаторе, чтобы можно было перехватить ее стандартный выходной поток и стандартный поток ошибок. В стандартный выходной поток (с помощью cat) выдается содержимое журнала, полученного от программы **sendmaiI.** В данном случае я хочу возвращать в качестве содержимого Web-страницы результаты, переданные как в стандартный выходной поток, так и в стандартный поток ошибок, чтобы в PL/SQL-коде можно было проанализировать все сообщения об ошибках и т.п. Проще всего добиться этого, перенаправив оба потока во временный файл и выдав содержимое файла в стандартный выходной поток. В конце мы удаляем временные файлы и завершаем работу.

Теперь можно протестировать эту процедуру:

```
scott@ORA8I.WORLD> begin
  2     sm('tkyte6us.oracle.com',
  3        'tkyte6us.oracle.com',
  4         'testing',
  5         'hello world!');
  6  end;
  7  /
m1.EMAIL.tkyte@us.oracle.com
tkyte@us.oracle.com
```

В результате выдано значение переменной среды **QUERY_STRING** и адрес получателя, и, поскольку других сообщений (об ошибках) не выдано, мы знаем, что сообщение электронной почты отправлено.

Итак, этот небольшой пример показывает, как использовать средства протокола **UTL_HTTP** в целях, для которых они изначально не предназначены, — для выполнения команд операционной системы из PL/SQL-процедур. Для этого необходимо настроить специализированный Web-сервер на IP-адресе **127.0.0.1,** сконфигурировать на нем каталог для cgi-программ и поместить в него команды, которые будут выполнять в нем разработчики PL/SQL-процедур. В результате получается безопасный способ выполнения аналога команды **host** утилиты SQL*Plus в языке PL/SQL.

Улучшенная версия пакета UTL_HTTP

При наличии класса **SocketType,** который представлен в разделе, посвященном пакету **UTL_TCP** (или при наличии хотя бы самого пакета **UTL_TCP**), а также знании особенностей протокола **HTTP**, можно создать улучшенную версию пакета **UTL_HTTP**. Мы назовем нашу реализацию **HTTP_PKG.** Она будет поддерживать прежние интерфейсы пакета **UTL_HTTP,** функции **REQUEST** и **REQUEST_PIECES**, и, кроме того, обеспечит следующее.

- **Получение HTTP-заголовков** для **каждого запроса.** Эти заголовки содержат полезную информацию, в частности состояние запроса (например, **200 OK, 404** Not **Found** и так далее), имя сервера, предоставившего URL, тип возвращенного содержимого, ключики и т.д.
- **Получение результатов в виде объекта типа CLOB или BLOB.** Это позволит выбирать из PL/SQL большие PDF-файлы в таблицу, индексированную с помощью компонента interMedia; выбирать простой текст с другой страницы или получать двоичные данные, возвращаемые Web-сервером.
- **Выполнение запросов HEAD для документа.** Это позволит проверить, например, изменился ли документ, который вы загружали на прошлой неделе.
- **Кодирование строк URL.** Например, при наличии в адресе URL пробела или тильды (~) их необходимо маскировать. Пакет будет маскировать все символы, которые должны быть замаскированы.
- **Посылка ключиков вместе с запросом.** Если пакет **HTTP_PKG** используется для обращения к Web-сайту, использующему аутентификацию на базе ключиков (cookie based authentication), эта возможность абсолютно необходима. Необходи-

мо выбрать (с помощью запроса **GET)** страницу регистрации, посылая в запросе имя пользователя и пароль. Затем необходимо обработать заголовки HTTP, возвращаемые для этой страницы, и извлечь соответствующий ключик. Это значение потребуется для идентификации при посылке последующих запросов.

- **Поддержка метода посылки POST, а не только GET.** Метод **GET** имеет ограничение на размер запроса, разное для различных Web-серверов. Обычно длина URL не должна превышать 1-2 Кбайт. Если необходимо передать большее значение, данные надо передавать с помощью метода **POST.** Метод **POST** не налагает ограничений на размер передаваемых данных.

Все это можно реализовать на языке PL/SQL с помощью типа **SocketType,** рассмотренного в разделе, посвященном пакету **UTL_TCP,** причем кода для этого придется написать сравнительно немного. Далее представлена спецификация нового пакета **HTTP_PKG.** Мы рассмотрим только спецификацию и несколько примеров использования этого пакета. В книге не приведен код, реализующий тело пакета **HTTP_PKG.** Этот код доступен и описан на сайте издательства Wrox, **(http://www.wrox.com)** — он занимает около 15 страниц печатного текста, поэтому и не приведен здесь. Спецификация пакета представлена ниже. Первые две функции, **REQUEST** и **REQUEST_PIECES,** аналогичны (за исключением поддержки SSL) одноименным функциям пакета **UTL_HTTP** в версиях 7.3.x, 8.0.x и 8.1.5, — мы даже возбуждаем исключительные ситуации с теми же именами:

```
tkyte@TKYTE816>create or replace package http_pkg
  2  as
  3     function request(url in varchar2,
  4                      proxy in varchar2 default NULL)
  5      return varchar2;
  6
  7     type html_pieces is table of varchar2(2000)
  8        index by binary_integer;
  9
 10     function request_pieces(url in varchar2,
 11                             max_pieces natural default 32767,
 12                             proxy in varchar2 default NULL)
 13     return html_pieces;
 14
 15     init_failed exception;
 16     request_failed exception;
```

Далее следует процедура **GET_URL,** вызывающая стандартную команду **GET** протокола HTTP на Web-сервере. Она принимает следующие параметры:

- **p_url** — адрес URL страницы, которую необходимо получить.
- **p_proxy — имя:<порт>** промежуточного сервера, который следует использовать. Пустое значение показывает, что промежуточный сервер использовать не нужно. Примеры: **p_proxy => 'www-proxy'** или **p_proxy => www-proxy:80'.**
- **p_status** — возвращаемый параметр. Это код статуса HTTP для запроса, возвращаемый Web-сервером. 200 означает успешное выполнение, 401 — что не прошла авторизация, и т.д.

- **p_status_txt** — возвращаемый параметр. Он содержит полный текст записи статуса HTTP. Например, он может получить значение **HTTP/1.0 200 OK.**
- **p_httpHeaders** — этот параметр можно задавать при вызове, а по завершении работы он будет содержать заголовки http с запрошенного адреса URL. Все установленные при вызове значения будут переданы Web-серверу как часть запроса. В этом же параметре по результатам вызова будут все заголовки, которые вернул Web-сервер. Этот параметр можно использовать для посылки ключиков или простой аутентификации, или для передачи необходимого заголовка http.
- **p_content** — временный объект типа **CLOB** или **BLOB** (в зависимости о того, какая версия перегруженной процедуры вызывается), который автоматически создается пакетом (вам его создавать не нужно). Это временный большой объект для сеанса. Для освобождения памяти этого объекта в любой момент можно вызвать процедуру **dbms_lob.freetemporary,** но можно и оставить его до конца сеанса - тогда объект будет освобожден автоматически.

```
19     procedure get_url(p_url       in varchar2,
20                       p_proxy     in varchar2 default NOLL,
21                       p_status          out number,
22                       p_status_txt      out varchar2,
23                       p_httpHeaders   in out CorrelatedArray,
24                       p_content    in out clob);
25
26
27     procedure get_url(p_url       in    varchar2,
28                       p_proxy     in    varchar2 default NOLL,
29                       p_status          out number,
30                       p_status_txt      out varchar2,
31                       p_httpHeaders   in out CorrelatedArray,
32                       p_content    in out blob);
```

Далее следует процедура **HEAD_URL**. Она выполняет стандартную команду **HEAD** протокола **HTTP** на Web-сервере. Входные и выходные параметры аналогичны одноименным параметрам рассмотренной ранее процедуры **get_url** (но содержимое страницы не выбирается). Эта процедура пригодится для определения того, существует ли документ, какой у него mime-тип и был ли он изменен, без загрузки содержимого документа:

```
34     procedure head_url(p_url       in varchar2,
35                        p_proxy     in varchar2 default NULL,
36                        p_status       out number,
37                        p_status_txt      out varchar2,
38                        p_httpHeaders   out CorrelatedArray);
```

Следующая функция, **URLencode**, используется при построении списка параметров GET или объектов типа **CLOB** для метода **POST**. Она используется для маскировки специальных символов в адресах **URL** (Например, URL не может содержать пробел, символ % и т.п.). При передаче строки **Hello World**, функция **urlencode** вернет **Hello%20World.**

```
40     function urlencode(p_str in varchar2) return varchar2;
```

Процедура **Add_A_Cookie** позволяет установить значение ключика для посылки Web-серверу. Надо передать только имя и значение ключика. Этой процедурой формируется соответствующий заголовок HTTP. Передаваемая в режиме in out переменная **p_httpHeaders** в дальнейшем передается процедурам **<Get|Head|Post>_url:**

```
42    procedure add_a_cookie
43              (p_name in varchar2,
44              p_value in varchar2,
45               p_httpHeaders in out CorrelatedArray);
```

Следующая процедура, **Set_Basic_Auth**, позволяет задать имя пользователя/пароль для доступа к защищенной странице. Эта процедура формирует соответствующий заголовок HTTP. Передаваемая в режиме **in out** переменная **p_httpHeaders** в дальнейшем передается процедурам **<Get|Head|Post>_url:**

```
47    procedure set_basic_auth
48              (p_username in varchar2,
49               p_password in varchar2,
50                p_httpHeaders in out CorrelatedArray);
```

Процедура **set_post_parameter** используется при задании адреса URL с большим (размером более 2000 байт или около того) набором входных параметров. Для больших запросов рекомендуется использовать метод **POST**. Эта процедура позволяет поочередно добавлять параметры к запросу **POST.** Запрос **POST** формируется в переданном параметре типа **CLOB:**

```
52    procedure set_post_parameter
53               (p_name in varchar2,
54               p_value in varchar2,
55               p_post_data in out clob,
56                p_urlencode in boolean default FALSE);
```

Следующие две процедуры аналогичны рассмотренным ранее процедурам **GET_URL,** но в них добавлен входной параметр **p_post_data. p_post_data** — это объект типа **CLOB**, созданный с помощью вызовов рассмотренной выше процедуры **set_post_parameter** Оставшиеся входные и выходные параметры имеют такое же назначение, что и в процедурах **GET_URL**:

```
58    procedure post_url
59              (p_url        in     varchar2,
60              p_post_data  in     clob,
61              p_proxy      in     varchar2 default NULL,
62               p_status           out number,
63               p_status_txt       out varchar2,
64             p_httpHeaders   in out CorrelatedArray,
65             p_content    in out clob) ;
66
67    procedure post_url
68          (p_url           in     varchar2,
69              p_post_data  in     clob,
70              p_proxy      in     varchar2 default NULL,
71               p_status           out number,
72               p_status_txt       out varchar2,
```

```
 73                p_httpHeaders   in out CorrelatedArray,
 74                 p_content    in out blob);
 75
 76
 77  end;
 78  /

Package created.
```

Вот и вся спецификация пакета; назначение задаваемых в нем процедур легко понять. Можно выполнять вызовы вроде **GET_URL** для получения страницы по указанному адресу URL. При этом используется метод **GET** протокола **HTTP** для сброса содержимого соответствующей Web-страницы во временный объект типа **BLOB** или **CLOB.** Можно вызвать **HEAD_URL** для получения заголовков соответствующей страницы. Так можно определить mime-тип, например, чтобы решить, использовать ли для получения страницы объект типа **CLOB** (для **text/html)** или типа **BLOB** (для **image/gif)-** Можно вызвать процедуру **POST_URL** для того, чтобы послать по указанному адресу большие объемы данных. Есть также вспомогательные процедуры для установки ключиков в заголовке, для кодирования имени пользователя и пароля при простой аутентификации и т.д.

Теперь, предполагая, что вы загрузили реализацию пакета **HTTP_PKG** (тело пакета состоит из примерно 500 строк кода на языке PL/SQL), можно попробовать его в действии. Я представлю несколько служебных процедур, которые пригодятся при тестировании. Процедура **Print_clob** выдает содержимое объекта типа **CLOB** с помощью пакета **DBMS_OUTPUT.** Процедура **Print_Headers** аналогичным образом выдает заголовки HTTP, выбранные ранее в переменную типа **CorrelatedArray** (это объектный тип, являющийся частью реализации пакета **HTTP_PKG).** Упоминаемая ниже процедура P представлена в разделе, посвященном пакету **DBMS_OUTPUT,** и позволяет выдавать длинные строки:

```
ops$tkyte@DEV816>create or replace procedure print_clob(p_clob in clob)
  2  as
  3      l_offset number default 1;
  4  begin
  5     loop
  6      exit when l_offset > dbms_lob.getlength(p_clob);
  7       dbms_output.put_line(dbms_lob.substr(p_clob, 255, l_offset));
  8       l_offset := l_offset + 255;
  9     end loop;
 10  end;
 11  /

ops$tkyte@DEV816> create or replace
  2 procedure print_headers(p_httpHeaders correlatedArray)
  3  as
  4  begin
  5      for i in 1 .. p_httpHeaders.vals.count loop
  6     p(initcap(p_httpHeaders.valsi).name)     || ': ' ||
  7                       p_httpHeaders.vals(i).value);
  8       end loop;
  9       p(chr(9));
```

```
 10  end;
 11  /
```

Теперь переходим к тесту:

```
ops$tkyte@DEV816> begin
  2       p(http_pkg.request('http://myserver/'));
  3   end;
  4   /
<HTML>
<HEAD>
<TITLE>Oracle Service Industries</TITLE>
</HEAD>
<FRAMESET COLS="130,*"
border=0>
<FRAME SRC="navtest.html" NAME="sidebar" frameborder=0>
<FRAME SRC="folder_home.html"
NAME="body" frameborder="0" marginheight="0" marginwidth="0">
</FRAMESET>

</BODY>
</HTML>

ops$tkyte@DEV816> declare
  2       pieces    http_pkg.html_pieces;
  3   begin
  4       pieces  :=
  5            http_pkg.request_pieces('http://www.oracle.com',
  6                                    proxy=>'www-proxy1');
  7
  8       for i in 1  .. pieces.count loop
  9            p(pieces(i));
 10       end loop;
 11   end;
 12   /
<head>
<title>Oracle Corporation</title>
<meta http-equiv="Content-Type" content="text/html;
. . .
```

Представленные выше фрагменты показывают, что процедуры **REQUEST и REQUEST_PIECES** в новом пакете работают так же, как и одноименные процедуры пакета **UTL_HTTP**. Они функционально идентичны. Теперь будем вызывать функцию **URLENCODE**, преобразующую "плохие" символы в управляющие последовательности в адресах URL и данных, передаваемых с помощью метода **POST**:

```
ops$tkyte@DEV816> select
  2   http_pkg.urlencode('A>C%{hello}\fadfasdfads~'[abc]:=$+''"')
  3   from dual;

HTTP_PKG.URLENCODE('A>C%{HELLO}\FADFASDFADS~'[ABC]:=$+''"')

------------------------------------------------------------

A%3EC%25%7Bhello%7D%5Cfadfasdfads%7E%60%5Babc%5D%3A%3D%24%2B%27%22
```

Как видите, специальные символы > и % преобразованы, соответственно, в %3E и %25, а остальные символы, такие как в слове hello, оставлены без изменений. Это позволяет безопасно использовать в запросах HTTP специальные символы.

Теперь проверим первую из новых процедур пакета для работы с адресами URL. Следующий вызов вернет начальную страницу Yahoo через промежуточный сервер, **www-proxyl**, (вам, конечно, придется подставить вместо него свой промежуточный сервер). Кроме того, мы будем просматривать возвращаемый статус HTTP: значение 200 свидетельствует об успешном выполнении. Мы также увидим заголовки HTTP, которые возвращает Yahoo. В них всегда присутствует значение mime-type — оно позволяет понять, какого рода данные можно ожидать. Наконец, мы выбираем и выдаем содержимое страницы:

```
ops$tkyte@DEV816> declare
  2      l_httpHeaders     correlatedArray;
  3      l_status    number;
  4     l_status_txt    varchar2(255);
  5      l_content     clob;
  6  begin
  7     http_pkg.get_url('http://www.yahoo.com/',
  8                      'www-proxyl',
  9                      l_status,
 10                      l_status_txt,
 11                      l_httpHeaders,
 12                      l_content);
 13
 14     p('The status was ' || l_status);
 15    p('The status text was ' || l_status_txt);
 16      print_headers(l_httpHeaders);
 17      print_clob(l_content);
 18  end;
 19  /
The status was 200
The status text was HTTP/1.0 200 OK

Date: Fri, 02 Feb 2001 19:13:26 GMT
Connection: close
Content-Type: text/html

<html><head><title>Yahoo!</title><base href=http://www.yahoo.com/><meta
http-equiv="PICS-Label"
```

Попытаемся выполнить запрос HEAD к начальной странице Web-сайта издательства Wrox и посмотрим, что нам удастся выяснить:

```
ops$tkyte@DEV816> declare
  2      l_httpHeaders     correlatedArray;
  3      l_status    number;
  4     l_status_txt    varchar2(255);
  5  begin
  6     http_pkg.head_url('http://www.wrox.com/',
  7                       'www-proxyl',
  8                       l_status,
  9                       l_status_txt,
```

```
 10                          l_httpHeaders);
 11
 12      p('The status was  ' || l_status);
 13     p('The status text was ' || l_status_txt);
 14       print_headers(l_httpHeaders);
 15 end;
 16 /
The status was 200
The status text was HTTP/1.1 200 OK

Server: Microsoft-IIS/5.0
Date: Fri, 02 Feb 2001 19:13:26 GMT
Connection: Keep-Alive
Content-Length: 1270
Content-Type: text/html
Set-Cookie: ASPSESSIONIDQQQQGGNQU=PNMNCIBACGKFLHGKLLBPEPMD; path=/
Cache-Control: private
```

По этим заголовкам понятно, что сайт Wrox работает на платформе Windows и обслуживается сервером Microsoft IIS. Понятно также, что используется технология ASP (это видно по присланному ключику). Если бы мы выбрали соответствующую страницу, ее содержимое имело бы длину 1270 байт.

Теперь хотелось бы проверить использование ключиков. Здесь я использую стандартную демонстрационную процедуру, входящую в состав **OAS** (Oracle Application Server — сервер приложений Oracle) и **iAS** (Oracle's Internet Application Server — сервер Internet-приложений Oracle), **cookiejar**, показывающую, как использовать ключики в Web-процедуре на языке PL/SQL. Программа **cookiejar** берет значение ключика, если оно установлено, увеличивает на единицу и возвращает клиенту. Мы увидим, как это происходит, с помощью нашего пакета. Мы пошлем серверу значение 55, предполагая, что он вернет нам 56:

```
ops$tkyte@DEV816> declare
  2      l_httpHeaders    correlatedArray;
  3       l_status         number;
  4       l_status_txt    varchar2(255);
  5       l_content        clob;
  6  begin
  7      http_pkg.add_a_cookie('COUNT', 55, l_httpHeaders);
  8      http_pkg.get_url
  9     ('http://myserver.acme.com/wa/webdemo/owa/cookiejar',
 10        null,
 11        l_status,
 12        l_status_txt,
 13        l_httpHeaders,
 14         l_content);
 15
 16     p('The status was ' || l_status ;
 17     p('The status text was ' || l_status_txt);
 18      print_headers(l_httpHeaders);
 19      print_clob(l_content);
 20 end;
```

```
 21  /
The status was 200
The status text was HTTP/1.0 200 OK

Content-Type: text/html
Date: Fri, 02 Feb 2001 19:14:48 GMT
Allow: GET, HEAD
Server: Oracle_Web_listener2.1/1.20in2
Set-Cookie: COUNT=56; expires=Saturday, 03-Feb-2001 22:14:48 GMT

<HTML>
<HEAD>
<TITLE>C is for Cookie</TITLE>
</HEAD>
<BODY>
<HR>
<IMG SRC="/ows-img/ows.gif">
<H1>C
is for Cookie</Hl>
<HR>
You have visited this page <STRONG>56</STRONG> times in the last 24
hours.
. . .
```

Как видите, значение ключика, 55, было передано, и сервер увеличил его до 56.Затем он вернул нам измененное значение вместе с датой его устаревания.

Теперь вы увидите, как обращаться к странице, для доступа к которой необходимо передать имя пользователя и пароль. Это делается так:

```
ops$tkyte@DEV816> declare
  2      l_httpHeaders   correlatedArray;
  3      l_status        number;
  4      l_status_txt    varchar2(255);
  5      l_content       clob;
  6  begin
  7     http_pkg.set_bas ic_auth('tkyte', 'tiger', l_httpheaders);
  8      http_pkg.get_url
  9       ('http://myserver.acme.com:80/wa/intranets/owa/print_user',
 10        null,
 11        l_status,
 12        l_status_txt,
 13        l_httpHeaders,
 14        l_content);
 15
 16     p('The status was ' || l_status);
 17     p('The status text was ' || l_status_txt);
 18      print_headers(l_httpHeaders);
 19      print_clob(l_content);
 20  end;
 21  /
The status was 200
The status text was HTTP/1.0 200 OK
```

```
Content-Type: text/html
Date: Fri, 02 Feb 2001 19:49:17 GMT
Allow: GET, HEAD
Server: Oracle_Web_listener2.1/1.20in2

remote user = tkyte
```

В данном случае я просто настроил DAD (Database Access Descriptor — дескриптор доступа к базе данных), в котором имя пользователя/пароль для доступа не хранится. Это означает, что Web-сервер ожидает, что в запросе будет указано соответствующее имя пользователя и пароль. Я передал их процедуре, которая выдала значение переменной среды **REMOTE_USER** (имя подключившегося удаленно пользователя) в PL/SQL.

Наконец, я продемонстрирую передачу данных по методу **POST.** Снова воспользуемся адресом службы Yahoo. Служба Yahoo позволяет получить информацию о курсах акций в формате, пригодном для электронных таблиц. Поскольку список наименований акций (stock symbols), курсы которых вас могут заинтересовать, может оказаться весьма большим, эти данные стоит передавать методом **POST.** Вот пример, демонстрирующий получение несколько курсов акций со службы Yahoo по протоколу HTTP. Данные возвращаются в формате CSV (Comma Separated Values — значения через запятую), который легко проанализировать и загрузить, например, в таблицу:

```
ops$tkyte@DEV816> declare
  2      l_httpHeaders   correlatedArray;
  3      l_status        number;
  4      l_status_txt    varchar2(255);
  5      l_content       clob;
  6      l_post          clob;
  7  begin
  8      http_pkg.set_post_parameter('symbols','orcl ^IXID ^DJI ^SPC',
  9                                  l_post, TRUE);
 10      http_pkg.set_post_parameter('format', 'sl1dlt1c1ohgv',
 11                                  l_post, TRUE);
 12      http_pkg.set_post_parameter('ext', '.csv',
 13                                  l_post, TRUE);
 14      http_pkg.post_url('http://quote.yahoo.com/download/quotes.csv',
 15                        l_post,
 16                        'www-proxy',
 17                        l_status,
 18                        l_status_txt,
 19                        l_httpHeaders,
 20                        l_content);
 21
 22     p('The status was ' || l_status);
 23      p('The status text was ' || l_status_txt);
 24      print_headers(l_httpHeaders);
 25      print_clob(l_content);
 26 end;
 27  /
The status was 200
The status text was HTTP/1.0 200 OK
```

```
Date: Fri, 02 Feb 2001 19:49:18 GMT
Cache-Control: private
Connection: close
Content-Type: application/octet-stream

"ORCL",28.1875,"2/2/2001","2:34PM",-
1.875,29.9375,30.0625,28.0625,26479100
"^IXID",1620.60,"2/2/2001","2:49PM",-45.21,1664.55,1667.46,1620.40,N/A
"^DJI",10899.33,"2/2/2001","2:49PM",-84.30,10982.71,11022.78,10888.01,N/A
"^SPC",1355.17,"2/2/2001","2:49PM",-18.30,1373.53,1376.16,1354.21,N/A
```

Резюме

В этом разделе мы рассмотрели использование стандартного пакета **UTL_HTTP.** Вы увидели, как, подойдя творчески, можно использовать средства пакета **UTL_HTTP** не только для получения данных с Web-серверов, но и для реализации в PL/SQL аналога команды **HOST.** С помощью нескольких строк кода мы обеспечили возможность посылки сообщений электронной почты в любой версии сервера Oracle, начиная с 7.3, с использованием пакетов **UTL_FILE** и **UTL_HTTP.**

Затем мы выяснили, как использовать средства пакета **UTL_HTTP** поверх протокола SSL — это недостаточно подробно описано в руководстве по стандартным пакетам, поставляемом корпорацией Oracle (да и вообще нигде толком не описано). Вы научились обращаться из PL/SQL-процедур к Web-сайтам, поддерживающим протокол SSL, используя средства Oracle Wallet Manager.

Кроме того, вы узнали, как улучшить и без того хороший пакет. Если в разделе, посвященном пакету **DBMS_OBFUSCATION_TOOLKIT,** мы создали оболочку для более простого и гибкого использования стандартного пакета, то здесь мы полностью заново реализовали аналогичный пакет, предоставив дополнительные, ранее никак не поддерживаемые возможности. За это в нашей реализации пришлось пожертвовать поддержкой протокола SSL, но полученный пакет и так пригодится во многих случаях.

Можно пойти дальше, и с помощью языка Java или внешних процедур на языке C добавить полную поддержку протокола SSL. Это можно сделать также с помощью множества общедоступных и выпускаемых сторонними производителями классов и библиотек.

Пакет UTL_RAW

Пакет UTL_RAW поставляется вместе с сервером Oracle, начиная с версии 7.1.6. Этот пакет утилит первоначально создавался командой разработчиков процедурного шлюза Trocedural Gateway) для доступа к данным мейнфреймов и преобразованию их в текстовый вид, впоследствии же был расширен разработчиками средств репликации. Он содержит четыре часто используемые функции (вы неоднократно встречались с ними по ходу изложения). Я опишу только эти четыре функции, поскольку именно они наиболее полезны. В пакете есть и другие функции (еще тринадцать, если быть точным), но я не буду их здесь описывать. Подробнее о них можно прочитать в руководстве *Supplied PL/SQL Packages Reference*.

Предлагаю вашему вниманию следующие четыре функции:

- CAST_TO_VARCHAR2. Приводит данные типа RAW к типу VARCHAR2.
- CAST_TO_RAW. Приводит данные типа VARCHAR2 к типу RAW.
- LENGTH. Возвращает длину переменной типа RAW.
- SUBSTR. Возвращает подстроку переменной типа RAW.

Эти функции используются при работе с двоичными данными. Мы использовали их в пакете CRYPT_PKG, который рассматривается в разделах приложения A, посвященных пакетам DBMS_OBFUSCATION_TOOLKIT, DBMS_LOB и UTL_TCP.

Начнем с функций CAST_. Они меняют значение поля типа переменной RAW на VARCHAR2, и наоборот. При этом содержащиеся в переменной данные не преобразуются. Обычно при присвоении значения типа RAW переменной типа VARCHAR2 длина значения переменной типа VARCHAR2 будет в два раза больше, чем значения типа

RAW, а само оно будет состоять из шестнадцатиричных цифр. Каждый байт данных типа **RAW** будет преобразован в шестнадцатиричный вид (это преобразование мы намеренно использовали в процедурах пакета DBMS_OBFUSCATION_TOOLKIT, например, чтобы выдавать зашифрованные данные на экран в шестнадцатиричном виде). В тех случаях, когда такое преобразование нежелательно, пригодится функция CAST_TO_VARCHAR2. Чтобы увидеть, что она делает, мы можем воспользоваться SQL-функцией **DUMP:**

```
tkyte@TKYTE816> create table t (r raw(10));
Table created.

tkyte@TKYTE816>insert into t values (utl_raw.cast_tq_raw('helloWorld'));
1 row created.

tkyte@TKYTE816>select dump(r) rl, dump(utl_raw.cast_to_varchar2(r)) rl
  2             from t;

Rl                                Rl
--------------------------------------- ------------------------
Typ=23 Len=10:                    Typ=1 Len=10:
104,101,108,108,111,87,111,114 104,101,108,108,111,87,111,114
,108,100                          ,108,100
```

Как видно по результату **DUMP,** единственное, что изменилось в данных — это значение атрибута **TYP.** Оно изменилось с 23 на 1. Если обратиться к руководству *Oracle Call interface Programmer's Guide* и посмотреть таблицу встроенных типов, можно обнаружить, что значение 23 соответствует типу **RAW** длинной до 2000 байт, а значение 1 — типу **VARCHAR2** длиной до 4000 байт. Функция **CAST_TO_VARCHAR2** просто меняет атрибут типа переменной, данные она не трогает. Именно это и требовалось, когда мы применяли функцию **DBMS_LOB.SUBSTR** к объекту типа **BFILE,** а **в** нем оказывался простой текст. Требовалось привести данные типа **RAW** к типу **VARCHAR2** так, чтобы они не преобразовались в шестнадцатиричный вид, т.е. надо было поменять только тип данных.

Функция **UTL_RAW.CAST_TO_RAW** выполняет обратное действие. При наличии данных типа **VARCHAR2,** которые необходимо обрабатывать как **RAW,** она позволит только поменять тип. Мы использовали эту функцию при реализации **SIMPLE_TCP_CLIENT** в разделе, посвященном стандартному пакету **UTL_SMTP.** PL/SQL-клиент посылает данные типа **VARCHAR2,** но на уровне языка Java необходимы массивы байтов (данные типа **RAW).** В языке PL/SQL выполнить соответствующее преобразование очень легко.

Еще две заслуживающие внимание функции — это **UTL_RAW.LENGTH** и **UTL_RAW.SUBSTR.** При передаче данных типа **RAW** стандартным функциям **LENGTH** и **SUBSTR** данные сначала неявно преобразуются в тип **VARCHAR2** (в шестнадцатиричное представление). К сожалению, встроенные функции не перегружены и не принимают данные типа **RAW:** они преобразуют их **в VARCHAR2.** Это означает, что возвращаемая функцией **LENGTH** длина будет вдвое больше фактической, а функция **SUBSTR** будет всегда возвращать шестнадцатиричную строку, причем придется также уточнять значения параметров, задающих смещение и длину подстроки. Функции пакета **UTL_RAW** восполняют этот пробел в функциональных возможностях. Они эквивалентны следующим SQL-операторам:

```
tkyte@TKYTE816> select utl_raw.length(r), length(r)/2 from t;

UTL_RAW.LENGTH(R) LENGTH(R)/2

               10          10

tkyte@TKYTE816> select utl_raw.substr(r,2,3) rl,
  2         hextoraw(substr(r,3,6))  r2
  3    from t
  4  /

R1          R2

656C6C      656C6C
```

Функции пакета **UTL_RAW** очень упрощают программирование. Вычисление смещений в байтах для стандартной функции **SUBSTR** выполняется сложнее, и при этом всегда нужно помнить о делении на два.

Пакет UTL_SMTP и отправка электронной почты

Пакет **UTL_SMTP,** впервые появившийся в версии Oracle 8.1.6, представляет собой интерфейс к протоколу Simple Mail Transfer Protocol (простой протокол передачи электронной почты). Он требует наличия в сети SMTP-сервера. Обычно в сети организаций работает по крайней мере один SMTP-сервер, поскольку протокол SMTP является наиболее популярным способом отправки сообщений электронной почты.

Пакет **UTL_SMTP** лучше всего подходит для отправки небольших текстовых сообщений из базы данных. Хотя он и поддерживает передачу вложений в любых форматах, но формировать документ с вложениями (multi-part document), например преобразовыватьдвоичные файлы-вложения в документы, закодированные с помощью средств mime, должен сам пользователь.

Здесь я возвращаюсь к примеру, который мы рассматривали в разделе, посвященном пакету **DBMS_JOB** (в этом примере использовались средства пакета **UTL_SMTP),** но добавлю к созданному в нем пакету новые функциональные возможности. Мы также рассмотрим альтернативы пакету **UTL_SMTP,** обеспечивающие больше возможностей, включая отправку вложений вместе с сообщениями. Поскольку SMTP — протокол низкого уровня, используем существующий общедоступный код для создания интерфейса более высокого уровня к протоколу SMTP; для этого понадобится очень немного кода.

UTL_SMTP - расширенный пример использования

В разделе, посвященном пакету **DBMS_JOB,** мы разобрались, как посылать сообщения электронной почты с помощью протокола **UTL_SMTP,** а также создавать види-

мость быстрой передачи, за счет асинхронного ее выполнения. В этом разделе мы также обеспечили включение передачи сообщений в транзакцию; при откате транзакции сообщение не отправлялось, а при фиксации — отправлялось. Поэтому я настоятельно рекомендую использовать пакет **DBMS_JOB** как дополнительный уровень интерфейса при создании средств передачи сообщений. Ранее в примере рассматривалась следующая процедура, работающая с пакетом **UTL_SMTP**:

```
tkyte@TKYTE816> create or replace
  2  PROCEDURE send_mail (p_sender      IN VARCHAR2,
  3                       p_recipient  IN VARCHAR2,
  4                       p_message    IN VARCHAR2)
  5  as
  6    l_mailhost VARCHAR2(255) := 'yourserver.acme.com';
  7    l_mail_conn utl_smtp.connection;
  8  BEGIN
  9    l_mail_conn :=utl_smtp.open_connection(l_mailhost, 25);
 10    utl_smtp.helo(l_mail_conn, l_mailhost);
 11     utl_smtp.mail(l_mail_conn, p_sender);
 12    utl_smtp.rcpt(l_mail_conn, p_recipient);
 13    utl_smtp.open_data(l_mail_conn);
 14     utl_smtp.write_data(l_mail_conn, p_message);
 15    utl_smtp.close_data(l_mail_conn);
 16    utl_smtp.quit(l_mail_conn);
 17 end;
 18 /
Procedure created.

tkyte@TKYTE816> begin
  2         send_mail('me@acme.com',
  3                   'you@acme.com',
  4                   'Hello Tom');
  5 end;
  6 /

PL/SQL procedure successfully completed.
```

Эта процедура работает, но ее возможности весьма ограничены. Она посылает сообщение только одному адресату, т.е. нельзя посылать копии (**CC** — Carbon Copy) или скрытые копии (**BCC** — Blind Carbon Copy), нельзя задать тему сообщения (сообщение всегда приходит с пустой строкой темы. Хотелось бы поддержать больше вариантов посылки сообщений.

Полное описание всех возможностей пакета **UTL_SMTP** требует глубокого изучения протокола SMTP, что выходит за рамки книги. Читатели, которых интересуют все возможности протокола SMTP, должны обратиться к рабочему документу RFC812, описанию протокола SMTP. Его можно получить по адресу **http://www.faqs.org/rfcs/rfc821.html.** Ниже я кратко опишу, как послать с помощью пакета **UTL_SMTP** сообщение, в котором поддерживаются:

- несколько адресатов **'To'**;
- несколько адресатов **'CC'**;

- несколько адресатов '**BCC**';
- размер текста до 32 Кбайт;
- строка темы;
- описательная строка 'from' (так, что в клиенте в поле From: указывается не только адрес электронной почты отправителя).

Ниже представлена спецификация PL/SQL- пакета, поддерживающего все эти возможности. В нем задается тип массива, позволяющий вызывающему давать список адресатов, а также спецификация PL/SQL-процедуры, которую мы будем реализовать:

```
tkyte@TKYTE816>create or replace package mail_pkg
  2  as
  3         type array is table of varchar2(255);
  4
  5         procedure send(p_sender_e-mail in varchar2,
  6                        p_from         in varchar2,
  7                        p_to           in array default array(),
  8                        p_cc           in array default array(),
  9                        p_bcc          in array default array(),
 10                        p_subject      in varchar2,
 11                        p_body         in long);
 12 end;
 13  /
Package created.
```

Тело пакета, реализующего перечисленные возможности, покажется достаточно простым, если понимать основы работы протокола **SMTP**, а также структуру сообщения электронной почты (как клиенты получают поля **From**, **To**, **CC** и т.д.)- Прежде чем изучать код, давайте разберемся, какую структуру может иметь сообщение. Рассмотрим следующий текст:

```
From: Oracle Database Account <me@acme.com>
Subject: This is subject
To: you@acme.com, us@acme.com
Cc: them@acme.com

Hello Tom, this is the mail you need
```

Именно это надо передать в качестве текста сообщения с помощью пакета **UTL_SMTP,** чтобы клиент электронной почты установил значения полей **From, Subject** и т.д. Для этого нет специальных команд в протоколе SMTP; заголовки помещаются в сообщение и отделяются от текста сообщения пустой строкой. Разобравшись в этом, послать сообщение со всеми необходимыми опциями будет очень легко. Для того чтобы послать сообщение нескольким адресатам, надо вызвать **UTL_SMTP.RCPT** несколько раз с различными адресами.

Итак, переходим к телу пакета. Оно начинается с объявления глобальных переменных. Разумеется, необходимо заменить в коде значение **g_mailhost,** указав имя сервера, к которому вы имеете доступ. Я указал формальное имя, **yourserver.acme.com:**

```
tkyte@TKYTE816>create or replace package body mail_pkg
  2  as
```

```
 3
 4  g_crlf       char(2)defaultchr(13)||chr(10);
 5  g_mail_conn   utl_smtp.connection;
 6  g_mailhost    varchar2(255)  := 'yourserver.acme.com';
 7
```

Далее идет служебная (не опубликованная) функция для передачи сообщения нескольким адресатам — она, по сути, формирует адреса для сообщения. В то же время, она формирует строки To: или CC:, которые посылаются как часть текста сообщения, и возвращает строку в соответствующем формате. Функция реализована отдельно, поскольку это действие придется выполнять отдельно для списков адресатов **To**, **CC** и **BCC**:

```
 8  function address_email(p_string in varchar2,
 9                         p_recipients in array)  return varchar2
10  is
11      l_recipients long;
12  begin
13     for i in 1 .. p_recipients.count
14     loop
15       utl_smtp.rcpt(g_mail_conn, p_recipients(i));
16        if (l_recipients is null)
17        then
18           l_recipients :=p_string || p_recipients(i) ;
19        else
20          l_recipients := l_recipients || ', ' || p_recipients(i);
21        end if;
22     end loop;
23     return l_recipients;
24  end;
25
26
```

Теперь рассмотрим функцию, входящую в спецификацию, — с ее помощью пользователи и будут посылать сообщения. Она начинается со служебной процедуры **writeData**, упрощающей передачу заголовков сообщения (записей **To**:, **From**:, **Subject**:). Если запись заголовка не пуста, процедура будет использовать для ее передачи соответствующий вызов пакета **UTL_SMTP**, добавляя необходимую метку конца строки (возврат каретки/перевод строки):

```
27  procedure send(p_sender_email in varchar2,
28                 p_from        in varchar2 default NOLL,
29                 p_to          in array default array(),
30                 p_cc          in array default array(),
31                 p_bcc         in array default array(),
32                 p_subject     in varchar2 default NOLL,
33                 p_body        in long  default NOLL)
34  is
35      l_to_list   long;
36      l_cc_list   long;
37      l_bcc_list  long;
38      l_date      varchar2(255) default
39                 to_char(SYSDATE, 'ddMonyyhh24:mi:ss');
```

```
40
41     procedure writeData(p_text in varchar2)
42      as
43      begin
44         if (p_text is not null)
45          then
46             utl_smtp.write_data(g_mail_conn, p_text || g_crlf);
47          end if;
48      end;
```

Теперь все готово для передачи сообщения. Эта часть мало отличается от простейшей процедуры, с которой мы начали. Она начинается точно так же — с подключения к SMTP-серверу и организации сеанса:

```
49  begin
50      g_mail_conn :=utl_smtp.open_connection(g_mailhost, 25);
51
52      utl_smtp.helo(g_mail_conn, g_roailhost);
53      utl_smtp.mail(g_mail_conn, p_sender_email);
54
```

Вот тут начинаются отличия — вместо однократного вызова **UTL_SMTP.RCPT** мы вызываем функцию **address_email**, которая (потенциально) может вызвать эту процедуру несколько раз, создавая попутно списки **To**: и **CC**:. Создается также список **BCC**:, но мы не буем включать его в сообщение (мы не хотим, чтобы адресаты видели этот список)

```
55      l_to_list   :=address_email('To:  ', p_to);
56      l_cc_list   :=address_email('Cc:  ', p_cc);
57      l_bcc_list  := address_email('Bcc:  ', p_bcc);
58
```

Теперь вызовем процедуру **OPEN_DATA**, чтобы передать текст сообщения. Код в строках 61-68 генерирует раздел заголовков. Строка 69 посылает текст сообщения (его содержимое), а строка 70 завершает передачу.

```
59      utl_smtp.open_data(g_mail_conn);
60
61     writeData('Date: ' || l_date);
62     writeData('From: ' || nvl( p_from, p_sender_email));
63     writeData('Subject: ' || nvl(p_subject, "(nosubject)'));
64
65      writeData(l_to_list);
66      writeData(l_cc_list);
67
68     utl_smtp.write_data(g_mail_conn, '' || g_crlf);
69     utl_smtp.write_data(g_mail_conn, p_body);
70     utl_smtp.close_data(g_mail_conn) ;
71     utl_smtp.quit (g_mail_conn) ;
72 end;
73
74
75 end;
```

```
 76 /
Package body created.
```

Работу этого пакета можно проверить следующим образом:

```
tkyte@TKYTE816> begin
  2     mail_pkg.send
  3     (p_sender_email => 'me@acme.com',
  4     p_from => 'Oracle Database Account <me@aone.com>',
  5     p_to=>mail_pkg.array('you@acme.com','us@acme.com' ) ,
  6    p_cc => mail_pkg.array(' them@acme.com ' ) ,
  7     p_bcc => mail_pkg.array('noone@dev.null'),
  8      p_subject => 'This is a subject',
  9      p_body => 'Hello Tom, this is the mail you need');
 10 end;
 11 /

PL/SQL procedure successfully completed.
```

В результате генерируется сообщение со следующим текстом:

```
Date: 13 May 01 12:33:22
From: Oracle Database Account <me@acme.com
Subject: This is a subject
To: you@acme.com, us@acme.com
Cc: them@acme.com

Hello Tom, this is the mail you need
```

Это сообщение посылается всем адресатам, включая **noone@dev.null,** хотя его адреса в тексте нет, поскольку он задавался в строке **BCC:.**

Этот пакет реализует наиболее типичные варианты использования средств стандартного пакета **UTL_SMTP.** Ранее я написал, что он позволяет посылать сообщения с вложениями и т.п., но для этого пользователю придется немало потрудиться. Необходимо:

- сформировать многосекционный документ, закодированный с помощью mime, что само по себе непросто;
- закодировать двоичные данные алгоритмом Base-64 (или любым аналогичным методом кодирования, таким так **uuencode, binhex** и т.д.).

Для этого потребуется (по скромной оценке) написать несколько сотен, если не тысяч, строк PL/SQL-кода. Вместо этого я рекомендую использовать уже написанный и очень надежный интерфейс JavaMail, который рассматривается далее.

Загрузка и использование интерфейса JavaMail

Чтобы использовать пакет **UTL_SMTP,** поддержка языка Java должна быть сконфигурирована в базе данных на сервере Oracle 8i. Дело в том, что пакет **UTL_SMTP** использует средства пакета **UTL_TCP,** а тот, в свою очередь, построен на основе Java-функций. (Помните: если поддержка языка Java в базе данных не включена, для передачи

простых сообщений по электронной почте можно использовать пакет **UTL_HTTP;** пример представлен в соответствующем разделе приложения А). Так что, если вы можете использовать пакет **UTL_SMTP,** обратитесь на сайт компании Sun и загрузите интерфейс JavaMail. Это позволит посылать из базы данных более сложные сообщения, включающие вложения. Дальнейшее изложение основано на результатах работы моего коллеги, Марка Пьермарини (Mark Piermarini), который помогает мне решить множество проблем, связанных с использованием языка Java.

Обратившись к странице **http://java.sun.com/products/javamal/index.html,** вы сможете загрузить с нее интерфейс JavaMail **API.** В загруженном архиве будет несколько сотен файлов, из которых понадобится только один. После загрузки интерфейса JavaMail API не забудьте также загрузить расширения JavaBeans™ Activation Framework, или JAF javax.activation). Это необходимо для работы пакета JavaMail.

После загрузки этих двух наборов файлов извлеките из архива JavaMail **API** файл **mail.jar,** а из архива JAF — файл **activation.jar.** Нам понадобятся только они, но вы можете прочитать соответствующую документацию и обнаружить множество других возможностей, которые мы не используем. Мы используем только часть интерфейса, связанную с передачей сообщений по электронной почте. Интерфейс включает также функции получения сообщений по протоколам **IMAP**, **POP** и другие средства.

С помощью **loadjava** загрузите в базу данных архивы **mail.jar** и **activation.jar,** но перед этим их необходимо упаковать в другом формате. Эти jar-файлы упакованы в формате, который не понимает интерпретатор байт-кода сервера. Необходимо распаковать ("unjar") и снова упаковать ("rejar") без сжатия или использовать средства типа **WinZip** для "переупаковки" их в виде zip-файла. В Windows 2000 я делал следующее:

- с помощью WinZip распаковывал содержимое файла **mail.jar** в каталог **c:\temp\mail;**
- с помощью WinZip создавал новый архив, **c:\temp\mail8i.zip;**
- помещал в этот новый архив содержимое **c:\temp\mail*.*,** в том числе подкаталоги.

То же самое я проделал для **activation.jar,** заменяя **mail** на **activation.** Теперь все готово для загрузки полученных zip-файлов (или jar-файлов, если вы заархивировали их в этом формате) в базу данных. Файлы должны загружаться в базу данных от имени пользователя **SYS,** поскольку они включают защищенные пакеты Java, которые обычным пользователям загружать не разрешено. Используем следующие команды **loadjava:**

```
loadjava -u sys/manager -o -r -v -f -noverify -synonym -g public
activation8i.zip
loadjava -u sys/manager -o -r -v -f -noverify -synonym -g public
mail8i.zip
```

Где:

- **u sys/manager** — идентификатор и пароль пользователя SYS. Защищенные пакеты должны загружаться от имени пользователя SYS.
- **o** — сокращение для -oci8. Я использую драйвер oci8. Вы можете использовать и thin-драйвер, но тогда команду придется соответственно изменить.

- **r** — сокращение для **-resolve.** Выполняется разрешение всех внешних ссылок в загруженных классах; это позволяет убедиться, что Java-классы после загрузки смогут работать.
- **v** — сокращение для **-verbose.** Эта опция обеспечивает выдачу сообщений по ходу работы программы **loadjava,** т.е. можно следить за ходом выполнения.
- **f** — сокращение для **-force.** При первой загрузке эту опцию указывать необязательно, но не помешает. Если при выполнении команды **loadjava** произойдет ошибка, можно будет исправить ее и повторить загрузку. В этом случае придется либо сначала удалить jar-файл из базы данных с помощью команды **dropjava,** либо указать опцию **-force.** Использование опции **-force** упрощает повторную загрузку.
- **noverify** — не проверять байт-код. Для выполнения этой опции необходимо получить привилегию **oracle.aurora.security.JServerPermission(Verifier).** Кроме того, эту опцию необходимо указывать вместе с опцией **-r**. Пользователь **SYS** имеет соответствующую привилегию. Опцию необходимо указывать, потому что средства проверки байт-кода сообщат о проблемах в файле **mail.jar,** а так мы сможем его успешно загрузить.
- **synonym** — создает общедоступные синонимы для классов. Поскольку мы не создаем от имени пользователя **SYS** пакет, использующий эти Java-классы, эта опция позволит нам увидеть классы, загруженные от имени **SYS.**
- **g public** — предоставляет привилегию **execute** на загруженные классы роли **PUBLIC.** Если это нежелательно, укажите после **-g** только имя пользователя, в схеме которого будут создавать процедуры отправки почты, например -g **UTILITY_ACCT.**

Подробнее о программе **loadjava** и ее опциях можно почитать в руководстве *Oracle8i Java Developers Guide*.

После загрузки этих пакетов все готово для создания хранимой процедуры на языке Java, фактически обеспечивающей передачу сообщений. Процедура, обращаясь к интерфейсу JavaMail **API,** позволит создать в **PL/SQL** связывающую функцию с такой спецификацией:

```
tkyte@TKYTE816>desc send
FUNCTION send RETURNS NUMBER
 Argument Name                  Type                    In/Out Default?

 P_FROM                         VARCHAR2                IN
 P_TO                           VARCHAR2                IN
 P_CC                           VARCHAR2                IN
 P_BCC                          VARCHAR2                IN
 P_SUBJECT                      VARCHAR2                IN
 P_BODY                         VARCHAR2                IN
 P_SMTP_HOST                    VARCHAR2                IN
 P_ATTACHMENT_DATA              BLOB                    IN
 P_ATTACHMENT_TYPE              VARCHAR2                IN
 P_ATTACHMENT_FILE_NAME         VARCHAR2                IN
```

Эта функция позволит использовать списки адресатов **CC** и **BCC** и посылать вложения. В качестве упражнения предлагаю читателю реализовать передачу массивов объектов типа **BLOB** или перезагрузку этой функции для работы с вложениями типа **CLOB** или **BFILE.**

Далее представлена соответствующая хранимая процедура на Java. Она использует базовые средства интерфейса класса JavaMail API и весьма проста. Мы снова не будем вникать в детали использования JavaMail **API** (этому можно посвятить отдельную книгу), рассмотрим только основы. Представленный ниже класс **mail** имеет один метод, **send.** Именно этот метод мы будем использовать для передачи сообщения. В представленной реализации он возвращает число 1 в случае успешной передачи и 0 — в противном случае. Эта реализация очень проста, можно предложить куда более сложную, поддерживающую вложение данных многих других типов **(CLOB, BFILE, LONG** и т.д.). Можно также выдавать вызывающему точную причину ошибки, полученную с сервера SMTP, например **invalid recipient, no transport.**

```
tkyte@TKYTE816>create or replace and compile
  2  java source named "mail"
  3  as
  4  import java.io.*;
  5  import java.sql.*;
  6  import java.util.Properties;
  7  import java.util.Date;
  8  import javax.activation.*;
  9  import javax.mail.*;
 10 import javax.mail.internet.*;
 11 import oracle.jdbc.driver.*;
 12 import oracle.sql.*;
 13
 14 public class mail
 15 {
 16   static String dftMime = "application/octet-stream";
 17   static String dftName = "filename.dat";
 18
 19   public static oracle.sql.NUMBER
 20                        send(String from,
 21                             String to,
 22                             String cc,
 23                             String bcc,
 24                             String subject,
 25                             String body,
 26                             String SMTPHost,
 27                            oracle.sql.BLOB attachmentData,
 28                             String attachmentType,
 29                             String attachmentFileName)
```

Представленный выше список аргументов соответствует спецификации вызова в PL/SQL, которая была показана ранее, — назначение аргументов вполне очевидно. Разъяснений требуют лишь два: **attachmentType** и **attachmentFileName. В** качестве **attachmentType** необходимо указывать тип MIME (Multi-purpose Internet Mail Extensions — многоцелевые расширения электронной почты для Internet), с которым вы могли встре-

чаться, например, в HTML-документах. MIME-типдля GIF-файла, например, — **image/gif,** mime-тип для обычного текстового документа — **text/plain,** а вложение в формате HTML должно иметь тип **text/html.** Значение **attachmentFileName** в данном случае — это не имя существующего файла ОС, который будет вкладываться в сообщение, а имя файла в самом вложении; оно будет выдано адресату сообщения в качестве имени вложения. Вложение передается процедуре фактически как данные типа **oracle.sql.BLOB**. Теперь переходим к телу функции. Оно начинается с установки свойству **mail.smtp.host** переданного значения имени SMTP-сервера — средства JavaMail API определяют по этому значению, к какому SMTP-серверу подключаться:

```
30   {
31    int rc = 0;
32
33    try
34    {
35    Properties props = System.getProperties();
36      props.put("mail.smtp.host", SMTPHost);
37      Message msg =
38          newMimeMessage(Session.getDefaultInstance(props, null));
39
```

Затем мы устанавливаем заголовки сообщения. Указывается, кто послал сообщение, кому его переслать, кому слать копии (**CC**) или скрытые копии (**BCC**), тема сообщения и дата отправки:

```
40    msg.setFrom(new InternetAddress(from));
41
42     if (to != null && to.length() > 0)
43       msg.setRecipients(Message.RecipientType.TO,
44                         InternetAddress.parse(to, false));
45
46     if (cc != null && cc.length() > 0)
47       msg.setRecipients(Message.RecipientType.CC,
48                         InternetAddress.parse(cc, false));
49
50     if (bcc != null && bcc.length() > 0)
51       msg.setRecipients(Message.RecipientType.BCC,
52                         InternetAddress.parse(bcc, false));
53
54     if (subject != null && subject.length() > 0)
55           msg.setSubject(subject);
56      elsemsg.setSubject("(no subject)");
57
58       msg.setSentDate(new Date());
59
```

Затем мы используем один из двух методов передачи сообщения. Если аргумент **attachmentData** — не пустой, будем кодировать сообщение с помощью **MIME**, стандарта, обеспечивающего передачу вложений и других многосекционных документов. Для этого мы создаем несколько частей **MIME**-текста; в данном случае две: текст сообщения и вложение. Строки 76-78 требуют дополнительного разъяснения. С их помощью

обеспечивается передача вложений, переданных как объект типа **BLOB**. Интерфейс **JavaMail** изначально не принимает данные типа **oracle.sql.BLOB** (ведь это универсальный интерфейс). Для отправки вложения в виде объекта типа **BLOB** необходимо задать метод получения его данных для интерфейса **JavaMail**. Для этого мы создаем специализированный класс **DataHandler**, интерфейс которого средства **JavaMail** смогут вызывать для получения данных вложения. Этот класс (**BLOBDataHandler**) реализован как вложенный:

```
60        if (attachmentData != null)
61        {
62         MimeBodyPart mbpl = new MimeBodyPart();
63     mbpl.setText((body     != null ? body : ""));
64         mbpl.setDisposition(Part.INLINE);
65
66         MimeBodyPart mbp2 = new MimeBodyPart();
67          String type =
68             (attachmentType != null ? attachmentType : dftMime);
69
70         String fileName = (attachmentFileName != null ?
71                            attachmentFileName : dftName);
72
73         mbp2.setDisposition(Part.ATTACHMENT);
74         mbp2.setFileName(fileName);
75
76          mbp2.setDataHandler(new
77             DataHandler(newBLOBDataSource(attachmentData, type))
78          );
79
80         MimeMultipart mp = new MimeMultipart();
81         mp.addBodyPart(mbpl);
82         mp.addBodyPart(mbp2);
83         msg.setContent(mp);
84        }
```

Если же вложения нет, текст сообщения передается очень просто — вызовом setText:

```
85        else
86        (
87         msg.setText((body != null ? body.....));
88        }
89        Transport.send(msg);
90        rc = 1;
91      } catch (Exception e)
92      {
93        e.printStackTrace();
94        rc = 0;
95      } finally
96      {
97        return new oracle.sql.NUMBER(rc);
98      }
99    }
100
```

Теперь рассмотрим вложенный класс, **BLOBDataSource**. Он реализует универсальный интерфейс для JavaMail API, позволяющий обращаться к данным типа **oracle.sql.BLOB**. Реализация этого класса очень проста:

```
101    // Вложенный класс, реализующий интерфейс DataSource.
102    static class BLOBDataSource implements DataSource
103    {
104      private BLOB   data;
105      private String type;
106
107      BLOBDataSource(BLOB data, String type)
108      {
109          this.type = type;
110          this.data = data;
111      }
112
113      public InputStreamgetInputStream() throws IOException
114      {
115        try
116        {
117          if(data == null)
118           throw new IOException("НеТ данных.");
119
120          return data.getBinaryStream();
121        } catch(SQLException e)
122        {
123          throw new
124          IOException("Не удалось получить входной поток двоичных
-> данных из BLOB.");
125        }
126      }
127
128      public OutputStream getOutputStream() throws IOException
129      {
130       throw new IOException("Не удалось это сделать.");
131      }
132
133      public String getContentType()
134      {
135        return type;
136      }
137
138      public String getName()
139      {
140        return "BLOBDataSource";
141      }
142    }
143  )
144  /

Java created.
```

Теперь, когда у нас есть Java-класс, созданный для связи с языком PL/SQL, необходимо создать функцию связи, сопоставляющую PL/SQL-типы Java-типам и связывающуюся с Java-классом. Она создается следующим образом:

```
tkyte@TKYTE816> create or replace function send(
  2      p_from                  in varchar2,
  3      P_to                    in varchar2,
  4      P_cc                    in varchar2,
  5      p_bcc                   in varchar2,
  6      p__subject              in varchar2,
  7      p_body                  in varchar2,
  8      p_smtp_host             in varchar2,
  9      p_attachment_data       in blob,
 10      p_attachment_type       in varchar2,
 11      p_attachment_file_name  in varchar2) return number
 12  as
 13  language java name 'mail.send(java.lang.String,
 14                                 java.lang.String,
 15                                 java.lang.String,
 16                                 java.lang.String,
 17                                 java.lang.String,
 18                                 java.lang.String,
 19                                 java.lang.String,
 20                                 oracle.sql.BLOB,
 21                                 java.lang.String,
 22                                 java.lang.String
 23                               ) return oracle.sql.NUMBER';
 24  /

Function created.
```

Последнее, что надо сделать, прежде чем использовать функцию, — убедиться, что пользователь (владелец класса **mail** и хранимой процедуры **send**) имеет достаточно привилегий для ее выполнения. Необходимые привилегии предоставляются следующим образом:

```
sys@TKYTE816> begin
  2     dbms_java.grant_permission(
  3       grantee => 'USER',
  4        permission_type => 'java.util.PropertyPermission',
  5       permission_name => '*',
  6       permission_action => 'read,write'
  7    );
  8     dbms_java.grant_permission(
  9       grantee => 'USER',
 10        permission_type => 'java.net.SocketPermission',
 11       permission_name => '*',
 12        permission_action => 'connect,resolve'
 13    );
 14  end;
 15  /

PL/SQL procedure successfully completed.
```

Обратите внимание: предоставляя привилегии для **java.net.SocketPermission,** я указал звездочку (*) в качестве **permission_name.** Это позволяет пользователю получать по имени IP-адрес и подключаться к хосту. Строго говоря, вы можете указать здесь только имя того SMTP-сервера, который будете использовать. Это будет минимально необходимая привилегия. Эта привилегия нужна, чтобы можно было получить IP-адрес по имени SMTP-хоста, а затем подключиться к нему. Другая привилегия, **java.util.PropertyPermission,** необходима для установки свойства сеанса **mail.smtp.host.**

Теперь все готово для проверки. Я использовал фрагмент кода из раздела, посвященного пакету **DBMS_LOB,** где рассматривалась процедура **load_a_file.** Изменив ее и таблицу **DEMO** так, чтобы в ней был столбец типа **BLOB** вместо **CLOB,** я загрузил в таблицу файл **mail8i.zip,** в котором находится использованный нами класс. Теперь я могу использовать такой PL/SQL-блок для передачи самому себе из базы данных этого класса как вложения в сообщении электронной почты:

```
tkyte@TKYTE816>set serveroutput on size 1000000
tkyte@TKYTE816> exec dbms_java.set_output(1000000)
tkyte@TKYTE816> declare
  2    ret_code number;
  3  begin
  4    for i in (select theBlob from demo)
  5    loop
  6      ret_code := send(
  7                    p_from=> 'me@acme.com',
  8                    p_to -> 'you@acme.com',
  9                     p_cc => NULL,
 10                    p_bcc => NULL,
 11                   p_subject => 'Use the attached Zip file',
 12                   p_body => 'to send email with attachments....',
 13                    p_smtp_host=> 'yourserver.acme.com',
 14                    p_attachment_data => i.theBlob,
 15                   p_attachment_type => 'application/winzip',
 16                     p_attachment_file_name => 'mail8i.zip');
 17     if ret_code = 1 then
 18      dbms_output.put_line ('Сообщение успешно послано...');
 19     else
 20      dbms_output.put_line ('Сообщение послать не удалось...');
 21     end if;
 22    end loop;
 23 end;
 24  /
Successfully sent message...

PL/SQL procedure successfully completed.
```

При проверке, несомненно, необходимо выполнить команду **set serverouput on** и вызвать процедуру **DBMS_JAVA.SET_OUTPUT**. Причина в том, что сообщения об исключительных ситуациях в хранимых процедурах на языке Java выдаются в **System.out** и по умолчанию попадают в файл трассировки на сервере. Если вы хотите получать сообщения об ошибках в сеансе SQL*Plus, необходимо выполнить эти два действия. Это очень поможет при отладке.

Резюме

В этом разделе мы кратко рассмотрели стандартный пакет **UTL_SMTP. Вы** узнали, как посылать сообщения нескольким адресатам, явно формировать заголовки **From:** и **Subject:.** Этого достаточно для удовлетворения требований большинства пользователей к средствами передачи сообщений электронной почты из базы данных. Пакет **UTL_SMTP** хорошо подходит для отправки простых, текстовых сообщений, но передача вложений или сложных, многосекционных сообщений выходит за пределы его возможностей (если только вы сами не сформируете тело сообщения). Если эти возможности необходимы, можно использовать средства JavaMail API. Поскольку компания Sun любезно предоставила необходимые для этого алгоритмы, мы просто воспользовались ее кодом. В этом разделе продемонстрировано не только, как посылать сообщения, но и какие побочные эффекты связаны с использованием Java в качестве альтернативного языка написания хранимых процедур. Теперь можно использовать общедоступный код и библиотеки классов. Можно обеспечить выполнение сервером многих действий, которые ранее были невозможны. Разработчики языка PL/SQL в корпорации Oracle сами используют этот метод. Пакет **UTL_TCP** в Oracle 8i построен на базе Java-функций.

Пакет UTL_TCP

В версии Oracle 8.1.6 появился пакет **UTL_TCP**. Он позволяет, работая в PL/SQL, устанавливать через сетевой сокет TCP/IP-соединение с любым сервером. Если известен протокол сервера, можно "общаться" с ним из PL/SQL. Например, зная протокол HTTP (Hyper Text Transfer Protocol — протокол передачи гипертекста), можно с помощью пакета **UTL_TCP** выполнить следующее:

```
test_jsock@DEV816> DECLARE
  2     c utl_tcp.connection; — TCP/IP-подключение к Web-серверу
  3     n number;
  4    buffer varchar2(255);
  5  BEGIN
  6    c :=utl_tcp.open_connection('proxy-server', 80);
  7    n :=utl_tcp.write_line(c, 'GEThttp://www.wrox.com/HTTP/1.0');
  8    n :=utl_tcp.write_line(c);
  9    BEGIN
 10      LOOP
 11         n:=utl_tcp.read_text(c, buffer, 255);
 12          dbms_output.put_line(buffer);
 13      END LOOP;
 14    EXCEPTION
 15      WHEN utl_tcp.end_of_input THEN
 16       NULL; — конец входных данных
 17    end;
 18    utl_tcp.close_connection(c);
 19  END;
 20  /
HTTP/1.1 200 OK
```

```
Date: Tue, 30 Jan 2001 11:33:50 GMT
Server: Apache/1.3.9 (Unix) mod_perl/1.21
ApacheJServ/1.1
Content-Type: text/html

<head>
<title>Oracle
Corporation</title>
```

Этот PL/SQL-блок позволил мне подключиться к серверу, в данном случае — промежуточному, с именем proxy-server. Через брандмауэр я попал в Intemet. Это произошло в строке 6. Затем я запросил Web-страницу, в строках 7 и 8. В строках с 10 по 13 мы получаем содержимое этой Web-страницы, включая все существенные заголовки HTTP (которые, кстати, стандартный пакет **UTL_HTTP** не позволяет получить). Затем пакет **UTL_TCP** возбуждает исключительную ситуацию **UTL_TCP.END_OF_INPUT**, страница получена, и мы выходим из цикла. После этого мы отключаемся.

Этот простой пример демонстрирует большую часть функциональных возможностей пакета **UTL_TCP**. Мы не вызываем функции типа **AVAILABLE**, информирующие о том, есть ли данные для получения. Мы не вызываем процедуру **FLUSH**, передающую все результаты, находящихся в буфере (буферизация не используется, поэтому такой вызов не нужен). Мы не использовали все возможные варианты вызовов **READ**, **WRITE** и **GET** для обмена данными через сокет, но представленный пример достаточно полно демонстрирует особенности использования пакета **UTL_TCP**.

Меня не всегда устраивает скорость работы подобных фрагментов кода. По моему опыту, пакет **UTL_TCP**, хотя и работает, но в данной версии (Oracle 8i) имеет недостаточную производительность. В версии 8.1.7.1 проблема низкой производительности решена (речь идет об исправлении ошибки **#1570972**).

Насколько же медленно работает подобный код? Представленный выше код для выборки документа размером 16 Кбайт требует от четырех до десяти секунд, в зависимости от платформы. Это, конечно, — медленно, особенно по сравнению с тем, что соответствующая функция пакета **UTL_HTTP** позволяет загрузить такой же документ за доли секунды. К сожалению, пакет **UTL_HTTP** не позволяет работать с ключиками, заголовками **HTTP**, двоичными данными, выполнять простейшую аутентификацию и т.п., поэтому альтернативные варианты часто оказываются полезными. Я думаю, можно сделать лучше. Для этого мы реализуем собственный пакет **UTL_TCP**. При этом будем использовать объектные типы, работа с которыми рассматривалась в главе 20. Мы реализуем в PL/SQL тип SocketType, частично на языке Java. В разделе, посвященном пакету **UTL_HTTP**, мы использовали этот же тип, **SocketType,** для создания более удобного пакета **UTL_HTTP**. Поскольку интерфейс создаваемого типа построен по аналогии с возможностями пакета **UTL_TCP**, когда в версии Oracle9i появится встроенная, более эффективная реализация пакета **UTL_TCP**, мы легко сможем изменить тело типа, используя обращения к пакету **UTL_TCP**, и отказаться от нынешней реализации на базе Java.

Тип SocketType

Объектный тип SocketType будет иметь следующую спецификацию:

```
tkyte@TKYTE816> create or replace type SocketType
  2  as object
  3  (
  4       — Приватные данные вместо передачи контекста
  5       — каждой процедуре, как при использовании
  6       — пакета UTL_FILE.
  7       g_sock        number,
  8
  9       — Функция, возвращающая CRLF. Для удобства.
 10      static function crlf return varchar2,
 11
 12       — Процедуры для передачи данных через сокет.
 13      member procedure send(p_data in varchar2),
 14      member procedure send(p_data in clob),
 15
 16      member procedure send_raw(p_data in raw),
 17      member procedure send_raw(p_data in blob),
 18
 19       — Функции для получения данных через сокет. При закрытии сокета
 20       — (получении eof) они возвращают Null. Будут ждать данных,
 21       — блокируя работу. Если это нежелательно, используйте
 22       — представленную ниже функцию PEEK, чтобы узнать, есть ли
-> данные для чтения.
 23      member function recv return varchar2,
 24      member function recv_raw return raw,
 25
 26      — Служебная функция. Читает данные, пока не обнаружит CRLF.
 27      — Может удалять CRLF, при желании пользователя (или не удалять,
-> по умолчанию).
 28      member function getline(p_remove_crlf in boolean default FALSE)
 29             return varchar2,
 30
 31       — Процедуры для подключения к хосту и отключения от него.
 32       — Важно не забывать отключаться, а то произойдет утечка
 33       — ресурсов и, рано или поздно, подключиться не удастся.
 34      member procedure initiate_connection(p_hostname in varchar2,
 35                                           p_portno   in number),
 36      member procedure close_connection,
 37
 38      — Функция, информирующая о том, сколько байтов (как минимум)
 39       — можно прочитать.
 40      member function peek return number
 41  );
 42  /

Type created.
```

Функциональные возможности этого типа во многом подобны предлагаемым пакетом **UTL_TCP**, да и интерфейс практически тот же. При желании тип можно реализовать на базе средств этого пакета. Мы, однако, собираемся реализовать его на базе другого пакета — **SIMPLE_TCP_CLIENT**. Это обычный пакет PL/SQL, на базе которого будет создан тип SocketType. Вот спецификация нашей версии пакета **UTL_TCP**:

```
tkyte@TKYTE816> CREATE OR REPLACE PACKAGE simple_tcp_client
  2  as
  3      — Функция для подключения к хосту. Возвращает "сокет",
  4      — который на самом деле представляет собой просто число.
  5      function connect_to(p_hostname in  varchar2,
  6                          p_portno   in  number) return number;
  7
  8      — Передача данных. Мы знаем только, как посылать данные типа
  9      — RAW. Вызывающие должны приводить данные типа VARCHAR2 к типу
 10      — RAW. На низком уровне, все проходящие через сокет данные —
-> байты.
 11
 12      procedure send(p_sock     in  number,
 13                     p_data     in  raw);
 14
 15      — Процедура recv предназначена для получения данных.
 16      — Если maxlength имеет значение -1, мы попытаемся получить
 17      — 4 Кбайт данных. Если maxlengthis имеет ЛЮБОЕ значение, кроме
 18      — -1, мы попытаемся прочитать все байты данных p_data. Другими
 19      — словами, я ограничиваю объем получаемых данных до 4 Кбайт,
-> если не сказано иначе.
 20      procedure recv(p_sock      in   number,
 21                     p_data      out  raw,
 22                     p_maxlength in   number default -1);
 23
 24      — Получает строку данных из входного сокета. Т.е. данные
 25      — вплоть до символа новой строки, \n.
 26      procedure getline(p_sock          in number,
 27                        p_data        out raw);
 28
 29
 30      — Обеспечивает отключение от сервера.
 31      procedure disconnect(p_sock     in number);
 32
 33      — Получает время сервера по Гринвичу (GMT) в формате yyyyMMdd
 34      HHmmss z procedure get_gmt(p_gmt out varchar2);
 35
 36      — Получает часовой пояс сервера. Полезно для некоторых
-> протоколов Internet.
 37      procedure get_timezone(p_timezone   out varchar2);
 38
 39      — Получает имя хоста, на котором работает ваш сервер. Это тоже
 40      — пригодится для некоторых протоколов Internet.
 41      procedure get_hostname(p_hostname   out varchar2);
 42
 43      — Возвращает количество байтов, которые можно прочитать.
 44      function peek(p_sock in number) return number;
 45
 46      — Кодирует данные типа RAW алгоритмом base64. Пригодится для
 47      — отправки вложений в сообщениях электронной почты или при
 48      — работе по протоколу HTTP, требующему скрывать имя
-> пользователя/пароль путем кодирования с помощью base64.
```

```
 49    procedure b64encode(p_data in raw, p_result out varchar2);
 50  end;
 51  /

Package created.
```

Поскольку ни одну из этих функций нельзя реализовать на PL/SQL, мы реализуем их на языке Java. Java-код, необходимый для этого, на удивление, невелик — занимает всего 94 строки. Будем использовать стандартный класс Socket языка Java и поддерживать небольшой массив сокетов, обеспечивающий в PL/SQL возможность поддерживать одновременно до десяти подключений. Если необходимо более десяти подключений, увеличьте размер массива **socketUsed** в представленном далее коде. Я пытался сделать код как можно проще и короче, предпочитая основную работу выполнять с помощью PL/SQL. Я представлю небольшой класс, который нам понадобится, и прокомментирую его:

```
tkyte@TKYTE816>set define off
tkyte@TKYTE816>CREATE or replace and compile JAVA SOURCE
  2  NAMED "jsock"
  3  AS
  4  import java.net.*;
  5  import java.io.*;
  6  import java.util.*;
  7  import java.text.*;
  8  import sun.misc.*;
  9
 10  public class jsock
 11  {
 12   static int         socketUsed[] = { 0,0,0,0,0,0,0,0,0,0 };
 13   static Socket      sockets[] = new Socket[socketUsed.length];
 14   static DateFormat   tzDateFormat = new SimpleDateFormat("z");
 15   static DateFormat   gmtDateFormat =
 16                         new SimpleDateFormat("yyyyMMdd HHmmss z");
 17  static BASE64Encoder encoder = new BASE64Encoder();
 18
```

В этом классе есть несколько статических переменных. Основные — это массивы **socketUsed** и sockets. При вызове функций из PL/SQL мы должны возвращать что-то, что можно будет передать в последующих вызовах для идентификации используемого сокета подключения. Мы не можем возвращать объекты Java-класса **Socket** в PL/SQL-подпрограммы, поэтому я использую массив, в котором они хранятся, и возвращаю в PL/SQL индекс этого массива. Метод **java_connect_to** просматривает массив **socketsUsed** в поисках пустого слота и выделяет этот слот подключению. Именно индекс в массиве **socketsUsed** и возвращается PL/SQL-подпрограммам. Эта особенность используется во всех остальных функциях работы с сокетами для получения доступа к фактическому Java-объекту, представляющему сокет.

Остальные статические переменные используются для повышения производительности. Мне необходимы объекты, форматирующие дату, и для того, чтобы не создавать их каждый раз при вызове **java_get_gmt или java_get_timezone,** я выделяю их один раз и в дальнейшем использую при необходимости. Наконец, объект для выполнения кодиро-

вания по алгоритму base 64. По той же причине, что и объекты для форматирования даты, я выделяю объект encoder.

Теперь рассмотрим функции для подключения к серверу по протоколу TCP/IP. Мы проходим в цикле по массиву socketUsed в поисках пустого слота (для которого значение socketUsed[I] отлично от 1).Если мы находим пустой слот, используется Java-класс Socket для подключения к указанному хосту/порту, и флаг socketUsed для этого слота массива устанавливается равным 1. В случае ошибки (нет свободного слота) возвращается значение -1; в случае успешного подключения — положительное число:

```
19 static public int java_connect_to(String p_hostname, int p_portno)
20 throws java.io.IOException
21 {
22 int   i;
23
24    for(i = 0; i < socketUsed.length && socketUsed[i] == 1; i++);
25     if (i < socketUsed.length)
26      {
27         sockets[i] = new Socket(p_hostname, p_portno);
28        socketUsed[i] = 1;
29      }
30    return i<socketUsed.length?i:-1;
31 }
32
33
```

Следующие две Java-функции вызываются чаще всего. Они отвечают за передачу и прием данных через подключенный TCP/IP-сокет. Функция **java_send_data** понятна: она получает выходной поток, соответствующий сокету, и выдает в него данные. Функция **java_recv_data** несколько сложнее. Она использует для возврата данных параметры типа OUT, отсюда, например, и объявление **int[] p_length**. Эта функция проверяет длину данных, переданных вызывающим, и, если она имеет значение -1, выделяет для чтения буфер размером 4 Кбайт; в противном случае она выделяет буфер указанного размера. Затем она пытается прочитать из сокета соответствующий объем данных. Реальный объем прочитанных данных (который будет меньше или равен затребованному) возвращается как результат в параметре **p_length**:

```
34 static public void java_send_data(int p_sock, byte[] p_data)
35 throws java.io.IOException
36 {
37   (sockets[p_sock].getOutputStream()).write(p_data);
38 }
39
40 static public void java_recv_data(int p_sock,
41                                   byte[][] p_data, int[] p_length)
42 throws java.io.IOException
43 {
44    p_data[0] = new byte[p_length[0] = -1 ? 4096:p_length[0] ];
45  p_length[0] = (socketstp_sock].getInputStream()).read(p_data[0]);
46 }
47
```

java_getline — это служебная функция. Многие протоколы Internet выдают результаты "построчно", и возможность получить строку текста всегда пригодится. Например, возвращаемые протоколом HTTP заголовки — это строки текста. Эта функция работает с помощью метода **DataInputStream.readLine**, возвращая сроку полученных данных и добавляя при этом символы новой строки, удаленные функцией **readLine** Если данные не получены возвращается пустое значение:

```
48 static public void java_getline(int p_sock, String[] p_data)
49 throws java.io.IOException
50 {
51     DataInputStream d =
52       new DataInputStream((sockets[p_sock].getInputStream()));
53     p_data[0] = d.readLine();
54    if (p_data[O] != null) p_data[0] += "\n";
55 }
56
```

Функция **java_disconnect** тоже очень проста. Она устанавливает флаг в массиве **socketUsed** для сокета снова в ноль, показывая, что этот слот в массиве сокетов можно использовать повторно, и закрывает сокет:

```
57 static public void java_disconnect(int p_sock)
58 throws java.io.IOException
59 {
60    socketUsed[p_sock] = 0;
61     (sockets[p_sock]).close();
62 }
63
```

С помощью функции **java_peek_sock** можно проверить, есть ли в сокете данные для чтения. Это пригодится в ситуациях, когда клиент не хочет ждать поступления данных, блокируя другие действия. Проверив, есть ли что читать, можно предвидеть, будет ли получение данных заблокировано или данные будут сразу возвращены:

```
64 static public int java_peek_sock(int p_sock)
65 throws java.io.IOException
66 {
67   return (sockets[p_sock].getInputStream()).available();
68 }
69
```

Осталось еще несколько функций, связанных со временем. Функция **java_get_timezone** используется для получения часового пояса, установленного на сервере базы данных. Она особенно пригодится, если данные типа DATE надо преобразовать из одного часового пояса в другой с помощью встроенной функции NEW_TIME или необходимо узнать, в каком часовом поясе работает сервер. Вторая функция, **java_get_gmt,** позволяет получить текущую дату и время на сервере по Гринвичу (GMT — Greenwich Mean Time):

```
70 static public void java_get_timezone(String[] p_timezone)
71 {
72   tzDateFormat.setTimeZone(TimeZone.getDefault());
73     p timezone[0] = tzDateFormat.fomat(new Date());
```

```
74 }
75
76
77 static public void java_get_gmt(String[] p_gmt)
78 {
79   gmtDateFormat.setTimeZone(TimeZone.getTimeZone("GMT"));
80    p_gmt[0] =gmtDateFormat.format(newDate());
81 }
82
```

Функция **b64encode** кодирует по алгоритму base64 переданную строку данных. Кодирование по алгоритму base64 — стандартный для Internet метод кодирования данных в виде последовательности 7-битовых символов ASCII для передачи по сетям. Мы будем использовать эту функцию, в частности, при реализации пакета для поддержки протокола HTTP, поскольку он поддерживает простейшую аутентификацию (используемую на многих Web-сайтах, требующих регистрации путем передачи имени пользователя и пароля).

```
83 static public void b64encode(byte[] p_data, String[] p_b64data)
84 {
85    p_b64data[0] = encoder.encode(p_data);
86 }
87
```

Последняя функция в этом классе возвращает имя хоста, на котором работает сервер базы данных. Некоторые протоколы Internet требуют передавать эту информацию (например, протокол SMTP — простой протокол передачи сообщений электронной почты):

```
88 static public void java_get_hostname(String[] p_hostname)
89 throws java.net.UnknownHostException
90 {
91   p_hostname[0] = (InetAddress.getLocalHost()).getHostName();
92 }
93
94 }
95 /

Java created.
```

Java-методы достаточно просты. Если вспомнить главу 19, для получения параметров в режиме **OUT** мы обязаны передавать массивы в Java-функцию. Поэтому большинство функций имеет вид:

```
40 static public void java_recv_data(int p_sock,
41                                    byte[][] p_data, int[] p_length)
```

Это позволяет возвращать значение в параметрах **p_data** и **p_length**. Теперь, при наличии Java-класса, можно создать тело пакета **SIMPLE_TCP_CLIENT**. Оно почти полностью состоит из интерфейсов к Java-методам:

```
tkyte@TKYTE816>CREATE OR REPLACE PACKAGE BODY simple_tcp_client
  2 as
  3
```

```
4      function connect_to(p_hostname in  varchar2,
5                          p_portno   in  number)  return number
6      as language java
7      name 'jsock.java_connect_to(java.lang.String, int) return int';
8
9
10     procedure send(p_sock in number, p_data in raw)
11     as language java
12     name 'jsock.java_send_data(int, byte[])';
13
14     procedure recv_i(p_sock      in number,
15                      p_data      out raw,
16                      p_maxlength in out number)
17     as language java
18     name 'jsock.java_recv_data(int, byte[][], int[])';
19
20     procedure recv(p_sock      in   number,
21                    p_data      out  raw,
22                    p_maxlength in   number default -1)
23     is
24         l_maxlength    number default p_maxlength;
25     begin
26        recv_i(p_sock, p_data, l_maxlength);
27         if (l_maxlength <> -1)
28         then
29            p_data := utl_raw.substr(p_data, 1, l_maxlength);
30         else
31             p_data := NULL;
32        end if;
33     end;
```

В данном случае есть две процедуры: **RECV_I** и **RECV. RECV_I** — это служебная процедура (суффикс **_I** в имени означает internal), которую нельзя вызвать непосредственно за пределами пакета. Она вызывается процедурой **RECV.** Процедура **RECV** обеспечивает "дружественный" интерфейс к процедуре **RECV_I:** она проверяет, были ли прочитаны данные из сокета и если — да, устанавливает соответствующую длину. Если вспомнить представленный ранее Java-код, мы выделяли буфер фиксированного размера в функции **RECV** и читали из сокета **не более** определенного количества байтов. В данном же случае необходимо изменить размер буфера так, чтобы он был равен количеству прочитанных байтов; для этого используется функция **UTL_RAW.SUBSTR.** Если же данные не прочитаны, возвращается **Null.**

```
34
35     procedure getline_i(p_sock      in number,
36                         p_data      out varchar2)
37     as language java
38     name 'jsock.java_getline(int, java.lang.String[])';
39
40     procedure getline(p_sock      in number,
41                       p_data        out raw)
42     as
```

```
43          l_data    long;
44      begin
45          getline_i(p_sock, l_data);
46          p_data := utl_raw.cast_to_raw(l_data);
47      end getline;
```

Аналогично рассмотренным выше процедурам **RECV_I/RECV**, **GETLINE_I** — внутренняя процедура, вызываемая только процедурой **GETLINE**. Внешний интерфейс PL/SQL представляет все данные как данные типа **RAW**, и функция **GETLINE** просто приводит данные типа **VARCHAR2** к типу **RAW**.

```
48
49      procedure disconnect(p_sock     in number)
50      as language java
51      name 'jsock.java_disconnect(int)';
52
53      procedure get_gmt(p_gmt   out varchar2)
54      as language java
55      name 'jsock.java_get_gmt(java.lang.String[])';
56
57      procedure get_timezone(p_timezone   out varchar2)
58      as language java
59      name 'jsock.java_get_timezone(java.lang.String[])';
60
61      procedure get_hostname(p_hostname   out varchar2)
62      as language java
63      name 'jsock.java_get_hostname(java.lang.String[])';
64
65      function peek(p_sock   in  number) return number
66      as language java
67      name 'jsock.java_peek_sock(int) return int';
68
69      procedure b64encode(p_data in raw, p_result out varchar2)
70      as language java
71     name 'jsock.b64encode(byte[], java.lang.String[])';
72 end;
73 /

Package body created.
```

Теперь можно протестировать процедуры, чтобы убедиться в их работоспособности:

```
tkyte@TKYTE816> declare
  2    l_hostname varchar2(255);
  3    l_gmt      varchar2(255);
  4    l_tz       varchar2(255);
  5  begin
  6    simple_tcp_client.get_hostname(l_hostname);
  7    simple_tcp_client.get_gmt(l_gmt);
  8    simple_tcp_client.get_timezone(l_tz);
  9
 10    dbms_output.put_line('hostname ' || l_hostname);
 11    dbms_output.put_line('gmt time ' || l_gmt);
```

```
 12     dbms_output.put_line('timezone ' || l_tz);
 13  end;
 14  /
hostname tkyte-dell
gmt time 20010131 213415 GMT
timezone EST

PL/SQL procedure successfully completed.
```

При использовании в базе данных компонентов этого пакета, связанных с протоколом TCP/IP, важно учитывать, что для этого необходимы соответствующие привилегии. Подробнее о пакете **DBMS_JAVA** и привилегиях, связанных с использованием Java, можно прочитать в разделе приложения А, посвященном пакету **DBMS_JAVA**. В данном случае необходимо выполнить:

```
sys@TKYTE816> begin
  2 dbms_java.grant_permission(
  3  grantee => 'TKYTE',
  4 permission_type => 'java.net.SocketPermission',
  5  permission_name => '*',
  6 permission_action=> 'connect,resolve');
  7  end;
  8  /

PL/SQL procedure successfully completed.
```

Подробнее о том, как и почему работает эта процедура, см. в разделе, посвященном пакету **DBMS_JAVA.** Если коротко, она позволяет пользователю **TKYTE** создавать подключения и получать по именам хостов IP-адреса (вот почему передано значение '*'). Если вы работаете с версией Oracle 8.1.5, пакета **DBMS_JAVAy** вас нет. **В** этой версии придется предоставить владельцу **jsock** привилегию **JAVASYSPRIV.** Учтите, что привилегия **JAVASYSPRIV** дает слишком широкие полномочия. Тогда как вызов **DBMS_JAVA.GRANT_PERMISSION** позволяет установить привилегию очень точно, привилегия **JAVASYSPRIV** предполагает широкий набор привилегий. Теперь, когда необходимые привилегии получены, можно создавать и тестировать тип **SocketType,** аналогично тому, как тестировался пакет **UTL_TCP**. Ниже представлено тело типа **SocketType.** Тело типа содержит очень немного кода — это просто оболочка для созданного пакета **SIMPLE_TCP_CLIENT.** Она скрывает "сокеты" от вызывающего:

```
tkyte@TKYTE816>create or replace type body SocketType
  2  as
  3
  4  static function crlf return varchar2
  5  is
  6  begin
  7     return chr(13)||chr(10);
  8  end;
  9
 10  member function peek return number
 11  is
 12  begin
 13     return simple_tcp_client.peek(g_sock);
```

```
14 end;
15
16
17 member procedure send(p_data in varchar2)
18 is
19 begin
20    simple_tcp_client.send(g_sock, utl_raw.cast_to_raw(p_data));
21 end;
22
23 member procedure send_raw(p_data in raw)
24 is
25 begin
26     simple_tcp_client.send(g_sock, p_data);
27 end;
28
29 member procedure send(p_data in clob)
30 is
31     l_offset number default 1;
32    l_length number default dbms_lob.getlength(p_data);
33     l_amt    number default 4096;
34 begin
35     loop
36         exit when l_offset > l_length;
37         simple_tcp_client.send(g_sock,
38                 utl_raw.cast_to_raw(
39                    dbms_lob.substr(p_data,l_amt,l_offset)));
40         l_offset  := l_offset + l_amt;
41     end loop;
42 end;
```

Процедура **SEND** — перегруженная и поддерживает параметры различных типов, в том числе данные типа **CLOB** произвольной длины. Для передачи она разбивает объект типа **CLOB** на 4-килобайтовые фрагменты. Представленная ниже процедура **SEND_RAW** работает аналогично, но с данными типа **BLOB**:

```
43
44 member procedure send_raw(p_data in blob)
45 is
46     l_offset number default 1;
47    l_length number default dbms_lob.getlength(p_data);
48     l_amt    number default 4096;
49 begin
50     loop
51         exit when l_offset > l_length;
52         simple_tcp_client.send(g_sock,
53                   dbms_lob.substr(p_data,l_amt,l_offset));
54         l_offset  := l_offset + l_amt;
55     end loop;
56 end;
57
58 member function recv return varchar2
59 is
```

```
 60      l_raw_data    raw(4096);
 61  begin
 62      simple_tcp_client.recv(g_sock, l_raw_data);
 63      return utl_raw.cast_to_varchar2(l_raw_data);
 64 end;
 65
 66
 67  member function recv_raw return raw
 68  is
 69      l_raw_data    raw(4096);
 70  begin
 71      simple_tcp_client.recv(g_sock, l_raw_data);
 72      return l_raw_data;
 73 end;
 74
 75  member function getline(p_remove_crlf in boolean default FALSE)
 76  return varchar2
 77  is
 78      l_raw_data    raw(4096);
 79  begin
 80      simple_tcp_client.getline(g_sock, l_raw_data);
 81
 82      if (p_remove_crlf) then
 83          return rtrim(
 84             utl_raw.cast_to_varchar2(l_raw_data), SocketType.crlf);
 85      else
 86          return utl_raw.cast_to_varchar2(l_raw_data);
 87     end if;
 88 end;
 89
 90  member procedure initiate_connection(p_hostname in varchar2,
 91                                       p_portno   in number)
 92  is
 93      l_data    varchar2(4069);
 94  begin
 95      — выполняется 10 попыток подключения и если ни одна из них не
 96      — увенчается успехом, распространяется исключительная ситуация
-> в вызывающую среду
 97      for i in 1 .. 10 loop
 98      begin
 99         g_sock := simple_tcp_client.connect_to(p_hostname, p_portno);
100          exit;
101      exception
102          when others then
103             if (i = 10) then raise; end if;
104     end;
105      end loop;
106 end;
```

Такое количество попыток выполняется для того, чтобы избежать проблем с сообщениями типа **"server busy"**. Это необязательно, но скрывает от вызывающего ошибки,

которые он получал бы слишком часто при обращении к загруженному Web-серверу или другой популярной службе.

```
107
108 member procedure close_connection
109  is
110 begin
111      simple_tcp_client.disconnect(g_sock);
112     g_sock := NULL;
113 end;
114
115 end;
116 /

Type body created.
```

Как видите, тело типа состоит в основном из вспомогательных подпрограмм, упрощающих использование средств пакета **SIMPLE_TCP_CLIENT**. Кроме того, мы инкапсулировали средства пакета **SIMPLE_TCP_CLIENT** в объектный тип. Используя тип **SocketType** вместо стандартного пакета **UTL_TCP**, можно реализовать блок кода для получения Web-страницы через промежуточный сервер:

```
tkyte@TKYTE816> declare
  2      s      SocketType  := SocketType(null);
  3      buffer varchar2(4096);
  4  BEGIN
  5     s.initiate_connection('proxy-server', 80);
  6    s.send('GET http://www.oracle.com/ HTTP/1.0'||SocketType.CRLF);
  7       s.send(SocketType.CRLF);
  8
  9      loop
 10         buffer := s.recv;
 11         exit when buffer is null;
 12        dbms_output.put_line(substr(buffer,1,255)) ;
 13      end loop;
 14     s.close_connection;
 15  END;
 16  /
HTTP/1.1 200 OK
Date: Thu, 01 Feb 2001 00:16:05 GMT
Server: Apache/1.3.9 (Unix) mod_perl/1.21
ApacheJServ/1.1
yyyyyyyyyy:close
Content-Type: text/html


<head>
<title>Oracle Corporation</title>
```

Это код как будто не отличается принципиально от кода, использующего средства пакета **UTL_TCP**, но показывает, как инкапсуляция пакетов в объектные типы создает приятное впечатление объектно-ориентированного программирования на PL/SQL. Тем, кто привык программировать на языках Java или C++, будет очень удобно работать с

подобного рода кодом, объявляя тип **SocketType** и вызывая затем его методы. Это намного удобнее, чем объявлять переменную-запись определенного типа и передавать ее затем в каждую подпрограмму, как приходится делать при использовании пакета UTL_TCP. Такая реализация — скорее объектно-ориентированная (в начале раздела была представлена чисто процедурная).

Резюме

В этом разделе мы рассмотрели новые возможности, предоставляемые пакетом **UTL_TCP.** Вы также познакомились с альтернативной реализацией этих возможностей на языке Java. На их основе мы сформировали новый объектный тип для языка PL/SQL, обеспечивающий все необходимые средства работы с сокетом **TCP/IP.** Вы увидели, как легко с помощью этого типа включить средства работы с сетью в PL/SQL-приложения (в разделе, посвященном пакету **UTL_HTTP,** было показано, как на основе этого объектного типа обеспечить полную поддержку протокола HTTP).

B

Поддержка, ошибки и сайт p2p.wrox.com

Одна из наиболее печальных ситуаций при чтении книг по программированию — когда взятый из книги фрагмент кода, который вы час набирали, не работает. Вы сто раз его проверяли, но потом обнаружили в тексте книги опечатку в имени переменной. Конечно, можно заклеймить позором авторов за то, что они были невнимательны и не протестировали код, редакторов — за то, что не справляются со своими обязанностями, корректоров — за то, что у них не орлиный глаз, но это не поможет решить проблему. Ошибки неистребимы.

Мы очень старались, чтобы ошибки в тираж книги не попали, но не можем обещать, что эта книга совершенно свободна от ошибок. Мы делаем все от нас зависящее, предоставляя оперативную поддержку и консультации экспертов, работавших над книгой, и надеемся, что в последующих изданиях найденные ошибки будут исправлены. Кроме того, мы собираемся консультировать читателей не только по вопросам, касающимся книги, но и по разработке приложений, через наши сетевые форумы, где можно задать вопросы автору, рецензентам и лучшим специалистам в соответствующей области.

В этом приложении описано, как:

- подключиться к работе форумов Programmer To Programmer™ на сайте http://p2p.wrox.com;
- найти (и послать) информацию об ошибках на нашем основном сайте, http://www.wrox.com;
- послать службе поддержки запрос или сообщить ваше мнение о книгах по электронной почте.

С помощью одной из этих процедур вы сможете быстро получить ответ по интересующей вас проблеме.

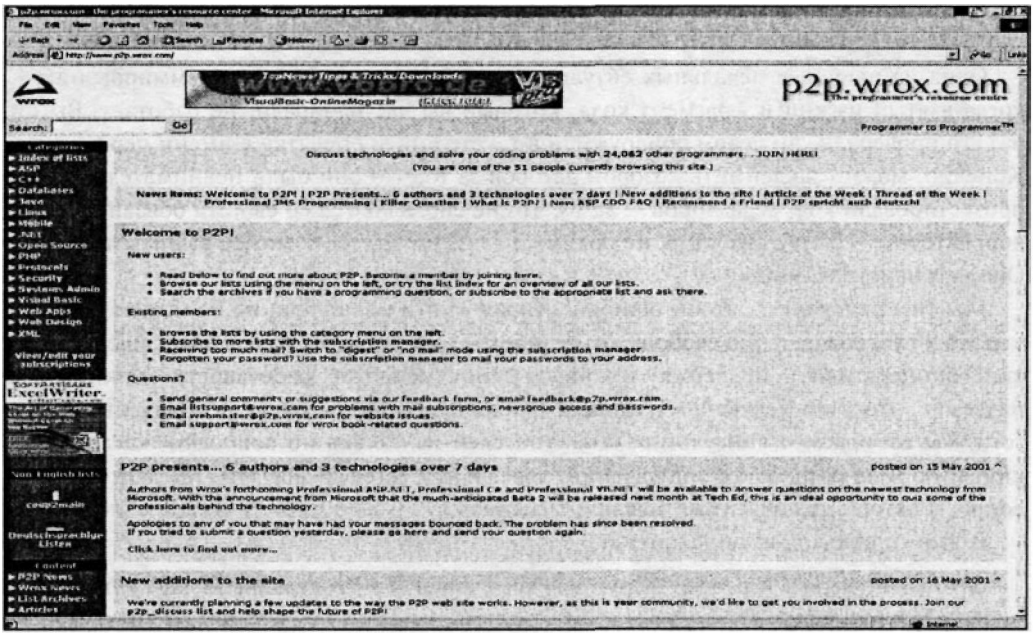
Форумы на сайте p2p.wrox.com

Подпишитесь на список рассылки по Oracle для получения помощи от автора и читателей. Наша система обеспечивает поддержку разработчиков "Программист — программисту" **(Programmer To Programmer™) с** помощью списков рассылки, форумов и дискуссионных групп, а также персональной (one-to-one) системы переписки по электронной почте, которую мы вскоре рассмотрим. Будьте уверены: ваш запрос будет рассмотрен не только специалистом службы поддержки, но и многочисленными авторами издательства Wrox и другими экспертами, читающими наши списки рассылки.

Как обратиться за поддержкой

Следуйте простой инструкции:

- Обратитесь на сайт **http://p2p.wrox.com** в любом браузере. Здесь вы найдете новости, связанные с работой службы P2P — информацию о создании новых списков рассылки, закрытии списков и т.д. Для этого:



- щелкните на кнопке **Databases** в левом столбце;
- выберите список рассылки **oracle;**
- если вы не подписаны на этот список рассылки, можно просматривать сообщения списка, не подписываясь на него, с помощью соответствующих кнопок;

- если вы решите подписаться, будет выдана форма, где вы укажете адрес электронной почты, имя и пароль (как минимум, из четырех алфавитно-цифровых символов). Выберите способ получения сообщений из списка и нажмите кнопку **Subscribe.**
- Поздравляем! Вы подписались на список рассылки **oracle.**

Почему эта система обеспечивает наилучшую поддержку

Подписываясь, вы можете выбрать получение сообщений по мере их поступления или раз в день. Можно также подписаться на еженедельный дайджест. Если у вас нет времени или возможности получать сообщения из списка рассылки, можете выполнять поиск в наших сетевых архивах. Вы можете искать по темам или по ключевым словам. Поскольку эти списки рассылки — контролируемые, можно быть уверенным в том, что вы быстро получите достоверную и точную информацию. Сообщения могут редактироваться или перемещаться в соответствующий раздел ведущим списка рассылки, что и делает данный ресурс наиболее эффективным. Ненужные и рекламные сообщения удаляются, а адреса электронной почты подписчиков защищаются уникальной системой Lyris от Web-роботов, которые автоматически перехватывают списки адресов дискуссионных групп. Любые вопросы, касающиеся подписки и списков, следует посылать по адресу **support@wrox.com.**

Поиск информации об ошибках на сайте www.wrox.com

В следующем разделе будет описано по шагам, как послать информацию о найденных ошибках и обратиться за помощью на наш сайт. Поэтому далее будут представлены подразделы, посвященные:

- поиску на Web-сайте списка известных ошибок;
- добавлению вашего сообщения об ошибке к списку уже известных ошибок.

Есть также подраздел, посвященный отправке сообщений в службу технической поддержки по электронной почте. В нем описано:

- какую информацию необходимо включить в сообщение;
- что происходит с сообщением после получения.

Поиск информации об ошибках на Web-сайте

Прежде чем обращаться с запросом, вы можете сэкономить время, найдя, возможно, ответ на сайте издательства по адресу **http://www.wrox.com.**

Для каждой опубликованной нами книги на сайте есть отдельная страница и отдельный список обнаруженных ошибок. Вы можете перейти на страницу, посвященную любой из книг, выбирая соответствующие ссылки в разделе **Books** левой навигационной панели. Для просмотра списка ошибок, найденных в этой книге, щелкните на ссылке **Book errata** в правой части панели с информацией о книге, ниже выходных данных.

Мы регулярно обновляем эти страницы, чтобы представить наиболее актуальную информацию об ошибках и опечатках.

Добавление информации об ошибке

Если вы хотите послать сообщение об ошибке, которое будет добавлено к соответствующей информации на Web-сайте, или обсудить касающуюся книги проблему непосредственно с экспертом, который досконально знает эту книгу, пошлите сообщение по адресу **support@wrox.com,** указав название книги и последние четыре цифры ISBN в поле темы сообщения. Щелчок на ссылке **submit errata** на странице с информацией об ошибках на сайте издательства приведет к отправке сообщения с помощью вашего стандартного почтового клиента. Сообщение об ошибке должно включать следующую информацию:

- название книги, последние четыре цифры ISBN и номер страницы, на которой она обнаружена, в поле темы;
- ваше имя, информацию о том, как с вами связаться, а также описание ошибки в тексте сообщения.

Мы не будем посылать вам сообщения, не относящиеся к обнаруженной ошибке. Подробная информация нужна нам, чтобы сэкономить свое и ваше время. Если понадобится заменить дискету или компакт-диск, мы сможем доставить его вам непосредственно. Присланные сообщения проходят по следующей цепочке поддержки:

Служба поддержки

Первым прочтет ваше сообщение один из сотрудников нашей службы сопровождения. В службе сопровождения имеется вся информация по наиболее часто задаваемым вопросам, и на типичные вопросы они ответят немедленно. Они дают ответы на общие вопросы, касающиеся книги и сайта издательства.

Редакция

Более сложные вопросы пересылаются научному редактору, который отвечает за книгу. У редакторов есть опыт работы с соответствующим продуктом или языком программирования, и они могут давать подробные технические ответы по тематике книги. После того как проблема решена, редактор размещает информацию об ошибке на Web-сайте.

Авторы

Наконец, в том маловероятном случае, когда редактор не может ответить на ваш вопрос, он пересылает его автору книги. Мы стараемся оградить автора от всего, что препятствует написанию книг, однако всегда с удовольствием пересылаем интересные вопросы. Все авторы, публикующие книги в издательстве Wrox, помогают отвечать на вопросы по своим книгам. Они посылают ответ читателю и редактору, и к этой информации получают доступ остальные читатели.

На какие вопросы мы не можем ответить

С ростом количества опубликованных книг и расширением спектра охваченных технологий службе поддержки приходится решать все больше проблем. Хотя мы стараемся ответить на все вопросы, касающиеся книги, мы не можем искать ошибки в ваших программах, созданных на основе описанных в книге методов. Поэтому, если, скажем, вам сильно понравились главы, посвященные работе с файлами, не надейтесь на всеобщее ликование после того, как написанная вами по мотивам этих глав программа удалила все файлы на сервере компании. А вот информацию о полезных программах, написанных с помощью книг нашего издательства, мы примем с удовольствием.

Предметный указатель

K

П

p

c

T

У

Ф

Х

Ц

Э

Я

Иностранные термины