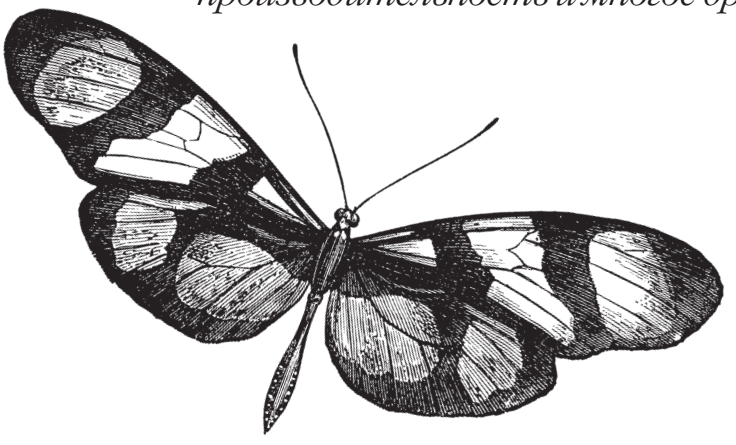
Безопасность, планирование, производительность и многое другое

Включая Oracle 10g Release 2



Oracle PL/SQL

для администраторов баз данных

Символ®

O'REILLY®

Арун Нанда и Стивен Фейерштейн

Oracle PL/SQL *for* DBAs

Arup Nanda and Steven Feuerstein

O'REILLY®

Oracle PL/SQL
для администраторов баз данных

Арун Нанда и Стивен Фейерштейн

Санкт-Петербург — Москва
2008

Аруп Нанда, Стивен Фейерштейн

Oracle PL/SQL для администраторов баз данных

Перевод П. Шера

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*О. Летаев*
Редактор	*Ю. Бочина*
Корректор	*С. Минин*
Верстка	*Д. Орлова*

Нанда А., Фейерштейн С.

Oracle PL/SQL для администраторов баз данных. – Пер. с англ. – СПб: Символ-Плюс, 2008. – 496 с., ил.

ISBN-10: 5-93286-101-0

ISBN-13: 978-5-93286-101-1

PL/SQL, мощнейший процедурный язык корпорации Oracle, является основой приложений, разрабатываемых на технологиях Oracle на протяжении последних 15 лет. Изначально PL/SQL предназначался только для разработчиков. Однако теперь он стал важнейшим инструментом администрирования баз данных, поскольку ответственность администраторов за производительность баз данных увеличилась, а границы между разработчиками и администраторами постепенно стираются.

«Oracle PL/SQL для администраторов баз данных» – первая книга, в которой язык PL/SQL рассматривается с точки зрения администрирования. Изложение ориентировано на версию Oracle 10g Release 2 и начинается с обзора PL/SQL, достаточного для знакомства администратора базы данных с основами этого языка и начала работы на нем. Далее подробно обсуждаются вопросы обеспечения безопасности, относящиеся к администрированию базы данных: шифрование (описаны как традиционные методы, так и новое прозрачное шифрование данных Oracle – TDE), контроль доступа на уровне строк (RLS), детальный аудит (FGA) и генерация случайных значений. Уделено внимание способам повышения производительности базы данных и запросов за счет применения курсоров и табличных функций. Рассматривается использование планировщика Oracle, позволяющего настроить регулярное выполнение таких заданий, как мониторинг базы данных и сбор статистики.

ISBN-10: 5-93286-101-0
ISBN-13: 978-5-93286-101-1
ISBN 0-596-00587-3 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 14.01.2008. Формат 70x100 1/16 . Печать офсетная.
Объем 31 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие

Во всем мире миллионы разработчиков приложений и администраторов баз данных используют продукты корпорации Oracle для создания сложных систем, управляющих огромными объемами данных. Значительная часть этих продуктов основана на PL/SQL – языке программирования, представляющем собой процедурное расширение Oracle-версии языка SQL (Structured Query Language – структурированный язык запросов) и использующемся для программирования в среде Oracle Developer.

Практически во всех новых продуктах, выпускаемых корпорацией Oracle, PL/SQL играет ключевую роль. Специалисты используют этот язык в различных областях программирования, в том числе:

- Для реализации основополагающей бизнес-логики на сервере Oracle с помощью хранимых PL/SQL-процедур и триггеров базы данных;
- Для формирования и обработки XML-документов внутри базы данных;
- Для связывания веб-страниц с базой данных Oracle;
- Для выполнения и автоматизации задач администрирования базы данных, начиная с реализации контроля доступа на уровне отдельных строк и заканчивая управлением сегментами отката в PL/SQL-программах.

PL/SQL разрабатывался на основе Ada[1] – языка программирования, созданного для Министерства обороны США. Ada – это язык высокого уровня, в котором особое внимание уделено абстракции данных, сокрытию информации и другим ключевым элементам современных технологий разработки. В результате такого выбора корпорации Oracle язык PL/SQL получился мощным средством, вобравшим в себя наиболее передовые элементы процедурных языков, такие как:

- Полный спектр типов данных, как числовых, так и строковых, включая такие сложные структуры данных, как записи (они подоб-

[1] Язык получил свое имя в честь Ады Лавлейс (Ada Lovelace), женщины-математика, которую многие считают первым программистом в истории человечества. Подробную информацию о языке Ada можно получить на вебсайте *http://www.adahome.com*.

ны строкам реляционной таблицы), коллекции (Oracle-версия массивов) и XMLType (для работы с XML-документами в Oracle и PL/SQL).

- Понятная и удобочитаемая блочная структура, благодаря которой сопровождение и внесение изменений в приложения PL/SQL становится простым и удобным.
- Операторы условного, итеративного и последовательного управления, в том числе оператор CASE и три различных вида циклов.
- Обработчики исключений, применяемые для событийной обработки ошибок.
- Допускающие повторное использование именованные элементы кода, такие как функции, процедуры, триггеры, объектные типы (родственники объектно-ориентированных классов) и пакеты (наборы связанных программ и переменных).

PL/SQL глубоко интегрирован в Oracle SQL: команды SQL можно выполнять непосредственно из процедурного кода, не прибегая к помощи какого-либо промежуточного API (Application Programming Interface – программный интерфейс приложений), подобного Java DataBase Connectivity (JDBC) или Open DataBase Connectivity (ODBC). Верно и обратное: вы можете вызывать свои PL/SQL-функции из операторов SQL.

Несомненно, основную часть пользователей PL/SQL составляют программисты, но пользуются им и многие администраторы баз данных. На самом деле владение PL/SQL жизненно необходимо администраторам баз данных Oracle.

PL/SQL для администраторов баз данных

Зачем администраторам баз данных нужен PL/SQL?

В самом общем виде ответ таков: именно администраторы баз данных отвечают за все, что находится (и исполняется) в их базах данных, включая код. Язык PL/SQL является важным рабочим инструментом, без помощи которого вы не сможете оценить безопасность, удобство эксплуатации и производительность ваших программ. Также вам не удастся воспользоваться преимуществами комплекса дополнительных функций, встроенных (обычно в виде поставляемых или встроенных пакетов) в базы данных Oracle и доступных *через* PL/SQL.

Давайте поговорим обо всем этом подробнее.

Обеспечение безопасности базы данных

Обеспечение безопасности всегда было ключевой задачей администратора базы данных, в последние же годы знание способов защиты базы данных и приложений приобретает все большее и большее значение. Многими элементами безопасности можно управлять непосредственно с помощью команд SQL и параметров конфигурации базы данных (на-

пример, установить пароли и определить роли и привилегии). Другие, более сложные методы защиты, такие как шифрование, контроль доступа на уровне строк, детальный аудит и генерация случайных значений, требуют применения PL/SQL. Эти методы детально рассматриваются в данной книге, при этом особое внимание уделяется использованию встроенных пакетов безопасности Oracle.

Оптимизация производительности

Разве не замечательно было бы, если бы все программисты а) хорошо разбирались в оптимизации операторов SQL, б) использовали бы самые последние разработки, повышающие производительность PL/SQL (такие как BULK COLLECT и FORALL), и в) не жалели бы времени на настройку своего кода?

И действительно, многие программисты уделяют значительное внимание эффективности работы своего кода. Другие же счастливы уже оттого, что он просто «работает». Но в конце концов код передается вам – администратору базы данных – для ввода в эксплуатацию. Поэтому (в зависимости от принятой именно в вашей компании концепции) может случиться, что именно вы будете отвечать за то, чтобы переданный разработчиком код не создал неполадок в реально работающей системе. По меньшей мере, вы должны быть способны дать необходимые рекомендации по вопросам производительности и предложить альтернативные подходы к реализации. Вы должны достаточно хорошо разбираться в PL/SQL и его последних версиях, чтобы суметь проанализировать код, выявить возможные «узкие места» и предложить разработчикам какие-то способы повышения производительности. При решении данной задачи вам будут особенно полезны главы об оптимизации курсоров и использовании табличных функций.

Эффективное использование возможностей Oracle

Когда-то администратору базы данных было достаточно «простого» SQL и команд конфигурации базы данных (работа велась в командной строке SQL*Plus или через графический интерфейс, подобный Oracle Enterprise Manager). Сегодня администратор базы данных должен, как минимум, уметь создавать PL/SQL-код для триггеров уровня схемы и базы данных, автоматизировать различные административные задачи при помощи динамического SQL (NDS) и других механизмов исполнения DDL, и активно использовать разнообразные новые возможности, предоставляемые во встроенных пакетах Oracle (начиная с потоков и заканчивая постановкой в очередь, тиражированием и использованием оптимизации на основе стоимости). И если PL/SQL окажется для вас камнем преткновения, то вы не сможете обеспечить достаточно эффективное администрирование базы данных для своей организации.

Воспитание новых разработчиков и администраторов баз данных

Многие делающие свои первые шаги в Oracle разработчики и администраторы баз данных не имеют достаточного опыта проектирования баз данных и оптимизации программного кода. Чем больше вы знаете о PL/SQL, – о том, как он работает и как писать хороший код, – тем более эффективно вы сможете способствовать профессиональному росту своих коллег. По мере повышения их квалификации уважение к вам будет расти, а ваша работа будет становиться все легче. Суть в том, что вы должны воспринимать владение PL/SQL как средство продвижения по карьерной лестнице администратора базы данных внутри своей компании и в своей отрасли в целом.

Об этой книге

Предложенные в этой книге материалы помогут вам в полной мере воспользоваться преимуществами важнейших для администраторов баз данных возможностей СУБД Oracle, основанных на PL/SQL.

Цель этой книги не в том, чтобы представить исчерпывающее описание языка Oracle PL/SQL. В главе 1 он будет рассмотрен достаточно подробно, в последующих же главах предполагается, что читатель обладает базовыми рабочими знаниями об этом языке программирования. Если вы не знакомы с языком PL/SQL, то советуем для начала прочитать книгу «Изучаем Oracle PL/SQL» («Learning Oracle PL/SQL»). В дальнейшем можно использовать в качестве справочника и руководства книгу «Программирование на Oracle PL/SQL», четвертое издание («Oracle PL/ SQL Programming» Fourth Edition). Этот 1200-страничный фолиант является классическим пособием по основам языка и его новым возможностям.

«Oracle PL/SQL для администраторов баз данных» состоит из восьми глав и приложения:

Глава 1 «Введение в PL/SQL» предлагает быстрый обзор языка PL/SQL, затрагивая все необходимые для администратора баз данных вопросы, начиная с основ блочной структуры PL/SQL, конструкции идентификаторов и объявлений данных в программах и заканчивая использованием управляющих операторов, обработкой ошибок, созданием процедур, функций, пакетов и триггеров в PL/SQL.

Глава 2 «Курсоры» описывает курсоры PL/SQL и способы повышения производительности базы данных за счет повторного использования курсоров, частичного разбора и частичного (мягкого) закрытия курсора, а также различных свойств явных и неявных курсоров. Кроме того, рассматривается применение типа данных `REF CURSOR`, массовой выборки, параметров курсоров и курсорных выражений.

Глава 3 «Табличные функции» исследует функции, которые могут использоваться как источники данных для запросов и которые часто используются в операциях ETL (Extraction, Transformation and Loading – извлечение, преобразование и загрузка). Табличные функции критически важны, когда необходимо реализовать сложную логику непосредственно в операторе SELECT, обычно для преобразования данных. В главе также рассказывается о том, как конвейерная обработка, распараллеливание и вложенное выполнение табличных функций позволяют достичь значительного повышения производительности.

Глава 4 «Шифрование и хеширование данных» поясняет, как можно использовать инструменты Oracle для создания базовой системы шифрования и управления ключами для защиты уязвимых данных. В главе рассматриваются операции шифрования, дешифрования, криптографического хеширования и использования MAC-кода (Message Authentication Code – код аутентификации сообщения) с подробным описанием использования встроенных пакетов DBMS_CRYPTO для Oracle Database 10*g* и DBMS_OBFUSCATION_TOOLKIT для Oracle9*i*. Также описывается новая возможность прозрачного шифрования данных (TDE – Transparent Data Encryption), появившаяся в версии Oracle Database 10*g* Release 2.

Глава 5 «Контроль доступа на уровне строк» рассказывает о том, как можно определить политики безопасности для таблиц баз данных с тем, чтобы ограничить подмножество строк этих таблиц, доступных для просмотра или изменения определенным пользователям. Используя пакет DBMS_RLS, вы также сможете предоставлять пользователям доступ к таблицам и представлениям только на чтение (в зависимости от представленных пользователями мандатов).

Глава 6 «Детальный аудит» показывает, как можно расширить стандартный аудит Oracle для сбора сведений об изменениях в базе данных и запросах. Используя пакет DBMS_FGA, вы сможете не только повысить безопасность, но и проанализировать отдельные примеры использования SQL и доступа к данным. В главе также описано, как FGA взаимодействует с ретроспективными запросами и триггерами Oracle.

Глава 7 «Генерирование случайных значений» рассматривает ситуации, в которых может потребоваться сгенерировать случайное значение (например, создание временных паролей или идентификаторов пользователей веб-сайта, формирование статистически корректных тестовых данных или создание ключей при построении инфраструктуры шифрования). Описывается использование встроенного пакета Oracle DBMS_RANDOM.

Глава 8 «Использование планировщика» посвящена использованию пакета DBMS_SCHEDULER (он появился в версии Oracle Database 10*g* и заменил старый пакет DBMS_JOB) при планировании заданий, которые должны выполняться через заданные промежутки времени (такие как сбор статистики, сбор информации о свободном пространстве или оповещение администратора базы данных о возникших проблемах).

Приложение А «Краткий справочник» содержит перечень спецификаций встроенных пакетов, описанных в книге, и представлений словаря данных, связанных с такими пакетами.

Используемые обозначения

В книге используются следующие условные обозначения:

курсив

Применяется при написании адресов URL и для выделения новых терминов.

Моноширинный шрифт

Применяется при написании имен файлов, атрибутов, функций, типов данных, пакетов и др., а также в примерах кода.

Моноширинный жирный шрифт

Обозначает вводимые пользователем данные в примерах, иллюстрирующих работу в диалоге. Также в некоторых примерах выделяет обсуждаемые операторы.

Моноширинный курсив

В некоторых примерах кода обозначает подставляемый фрагмент (например, имя файла).

ВЕРХНИЙ РЕГИСТР

В примерах кода обычно используется для обозначения ключевых слов PL/SQL.

нижний регистр

В примерах кода обычно используется для обозначения пользовательских элементов, таких как переменные, параметры и т. д.

знаки пунктуации

Должны вводиться именно так, как это указано в примерах кода.

отступ

В примерах кода служит для визуализации структуры, не является обязательным.

--

В примерах кода двойной дефис обозначает начало однострочного комментария, который продолжается до конца строки.

/* и */

В примерах кода эти символы определяют границы многострочного комментария, который может переходить с одной строки на другую.

.

В примерах кода и соответствующих фрагментах текста точка обозначает ссылку, отделяя имя объекта от имени компонента. Напри-

мер, точечная нотация используется для выбора полей записи и для объявлений внутри пакета.

[ ]

При описании синтаксиса в квадратные скобки заключаются необязательные элементы.

{ }

При описании синтаксиса в фигурные скобки заключается множество элементов, из которых следует выбрать только один.

|

При описании синтаксиса вертикальная черта разделяет элементы, заключенные в фигурные скобки, например `{TRUE | FALSE}`.

…

При описании синтаксиса многоточие обозначает повторяющиеся элементы. Кроме того, многоточие используется для того, чтобы показать, что были опущены не относящиеся к делу операторы или инструкции.

Обозначает совет, предложение или замечание. Например, указание на то, что какая-то конструкция присутствует только в определенных версиях.

Обозначает предупреждение. Например, мы хотим обратить внимание на то, что какая-то настройка может оказать негативное воздействие на систему.

Версии PL/SQL

Существует множество версий PL/SQL, и, возможно, вам как администратору базы придется работать с несколькими из них одновременно.

Базовой версией PL/SQL для нашей книги будет Oracle Database 10*g*. Однако при необходимости мы будем ссылаться на специальные возможности, введенные (или просто доступные) в других, более ранних версиях. Если какая-то функциональность напрямую зависит от версии, например, если ее можно использовать только в Oracle Database 10*g* Release 2, это будет особо отмечено в тексте.

Каждой версии базы данных Oracle соответствует собственная версия PL/SQL. Чем более свежую версию PL/SQL вы используете, тем больший спектр возможностей перед вами открыт. Пользователям PL/SQL следует всегда быть в курсе последних нововведений. Необходимо постоянно самосовершенствоваться, изучая новые возможности каждой версии, обдумывая, как можно было бы применить их в ваших приложениях, и определяя, есть ли среди предлагаемых новых приемов настолько полезные, что имеет смысл изменить уже существующие приложения, с тем чтобы воспользоваться новыми возможностями.

Основные элементы всех версий PL/SQL (прошлых и настоящей) представлены в табл. 1, которая дает самое общее представление о новых возможностях, предлагаемых в каждой версии.

Линия продуктов Oracle Developer также поставляется с собственной версией PL/SQL, которая обычно отстает от версии, доступной в самой СУБД Oracle. В этой главе (и в книге в целом) нас будет интересовать серверная реализация PL/SQL.

Таблица 1. Версии СУБД Oracle и соответствующие версии PL/SQL

Версия СУБД Oracle	Версия PL/SQL	Описание
6.0	1.0	Это исходная версия PL/SQL, которая использовалась главным образом как язык сценариев в SQL*Plus (еще не было возможности создания именованных, допускающих повторное использование и вызываемых программ) и как язык программирования в SQL*Forms 3.
7.0	2.0	Значительное усовершенствование PL/SQL 1.0. Была добавлена поддержка хранимых процедур, функций, пакетов, определяемых программистом записей, таблиц PL/SQL, а также много пакетов расширения.
7.1	2.1	Данная версия поддерживала определяемые программистом подтипы, разрешала использование хранимых функций внутри команд SQL и предлагала динамический SQL в пакете DBMS_SQL. В версии PL/SQL 2.1 наконец появилась возможность исполнять команды SQL DDL из программ PL/SQL.
7.3	2.3	Данная версия расширяла функциональность PL/SQL-таблиц, улучшала управление удаленными зависимостями, предоставляла возможности файлового ввода-вывода в PL/SQL с помощью пакета `UTL_FILE` и завершала реализацию курсорных переменных.
8.0	8.0	Номер новой версии отражал стремление корпорации Oracle к синхронизации номеров версий связанных продуктов. PL/SQL 8.0 – это версия PL/SQL, которая поддерживает новые возможности СУБД Oracle8, включая большие объекты (LOB), объектно-ориентированные проектирование и разработку, коллекции (VARRAY и вложенные таблицы) и опцию Oracle AQ (Advanced Queuing).
8.1	8.1	Версия PL/SQL для первой из серии «*i*» версии Oracle 8*i* предложила действительно впечатляющий набор дополнительных возможностей, включая новую версию динамического SQL, поддержку Java в базе данных, модель прав вызывающего, опцию полномочий на исполнение, автономные транзакции и высокопроизводительные «массовые» операторы DML и запросы.

Версия СУБД Oracle	Версия PL/SQL	Описание
9.1	9.1	Версия СУБД Oracle 9*i* Release 1 буквально наступала на пятки своей предшественнице. Она включала наследование объектных типов, табличные функции и курсорные выражения (что позволило распараллеливать исполнение функций PL/SQL), поддерживала многоуровневые коллекции, оператор и выражение `CASE`.
9.2	9.2	В версии СУБД Oracle 9*i* Release 2 основное внимание уделялось языку XML (Extensible Markup Language), а также были предоставлены многие другие дополнительные возможности, такие как ассоциативные массивы, для индексирования которых в дополнение к целым числам могли использоваться строки `VARCHAR2`, записеориентированные операторы DML (позволяющие, например, выполнить вставку с использованием записи) и множество усовершенствований `UTL_FILE` (для поддержки чтения/записи файлов из программы PL/SQL).
10.1	10.1	Версия Oracle Database 10*g* Release 1 была выпущена в 2004 году и посвящена поддержке распределенных вычислений, при этом особое внимание уделялось усовершенствованию и автоматизации управления базой данных. Очевидно, что для разработчиков PL/SQL важнейшими новыми возможностями были оптимизированный компилятор и предупреждения, выдаваемые в процессе компиляции.
10.2	10.2	Версия Oracle 10*g* Release 2, появившаяся осенью 2005, предложила разработчикам PL/SQL несколько новых возможностей, наиболее значимой из которых являлась поддержка синтаксиса препроцессора, делающая возможной условную компиляцию частей программы в зависимости от пользовательских логических выражений.

Обзор ресурсов по PL/SQL

Прежде чем перейти к своей основной задаче – описанию необходимых именно для администратора базы данных возможностей языка PL/SQL, мы предоставим нашему читателю описание основ PL/SQL. Однако существует множество других книг и ресурсов, которые помогут вам получить более глубокие знания по PL/ SQL.

В последующих разделах будет приведен краткий обзор таких ресурсов. Многие из них находятся в свободном доступе или распространяются за весьма небольшую плату. Знакомство с ними поможет вам усовершенствовать свое знание языка (а следовательно, и создаваемый код).

Серия O'Reilly, посвященная PL/SQL

Издаваемая на протяжении многих лет серия Oracle PL/SQL издательства O'Reilly включает в себя длинный список книг. Мы приведем перечень изданий, опубликованных на настоящий момент. Гораздо более полную информацию вы сможете найти в разделе Oracle веб-сайта O'Reilly (*http://oracle.oreilly.com*).

«Learning Oracle PL/SQL» (Изучаем Oracle PL/SQL), авторы Билл Прибыл (Bill Pribyl) и Стивен Фейерштейн (Steven Feuerstein)

Несколько неформальное знакомство с языком, идеальное как для новичков в программировании, так и для тех, кто знаком с каким-то другим языком. Особое внимание уделено разработке веб-приложений на PL/SQL.

«Oracle PL/SQL Programming» (Программирование на Oracle PL/SQL), автор Стивен Фейерштейн (Steven Feuerstein) с участием Билла Прибыла (Bill Pribyl)

Эта книга, лежащая на столе у большинства профессиональных PL/SQL-программистов и администраторов баз данных, на 1200 страницах охватывает все возможности языка PL/SQL. Четвертое издание описывает функциональность вплоть до версии Oracle Database 10*g* Release 2.

«Oracle PL/SQL for DBAs» (Oracle PL/SQL для администраторов баз данных), авторы Аруп Нанда (Arup Nanda) и Стивен Фейерштейн (Steven Feuerstein)

В книге, которую вы сейчас читаете, приводится краткий обзор всех возможностей языка PL/SQL, а углубленно рассматриваются темы, имеющие особое значение для администраторов баз данных, такие как курсоры, табличные функции, шифрование и хеширование данных, контроль доступа на уровне строк, детальный аудит, генерация случайных значений и использование планировщика. Книга включает и описание возможностей Oracle Database 10*g* Release 2.

«Oracle PL/SQL Best Practices» (Oracle PL/SQL. Лучшие практические методы), автор Стивен Фейерштейн (Steven Feuerstein)

Небольшая книга, описывающая более 100 приемов, которые помогут вам писать качественный PL/SQL-код. С читателем делится своим опытом специалист по PL/SQL. Изначально книга создавалась для СУБД Oracle8*i*, но практически все данные в ней рекомендации применимы и для более новых версий.

«Oracle PL/SQL Developer's Workbook» (Задачник для разработчика на Oracle PL/SQL), авторы Стивен Фейерштейн (Steven Feuerstein) и Эндрю Одеван (Andrew Odewahn)

Сборник вопросов и ответов для проверки понимания языка разработчиками на PL/SQL. Актуально для СУБД Oracle8*i*.

«Oracle Built-in Packages» (Встроенные пакеты Oracle), авторы Стивен Фейерштейн (Steven Feuerstein), Чарльз Дай (Charles Dye) и Джон Бересниевич (John Beresniewicz)

Справочник по встроенным пакетам, которые Oracle поставляет вместе с сервером базы данных. Применение этих пакетов позволяет упростить сложные задачи и решить невыполнимые. Эта книга соответствует версии Oracle8, но обсуждение встроенных пакетов все еще представляет интерес. Более актуальные данные о синтаксисе спецификации пакетов вы найдете в «Oracle in a Nutshell»[1] Рика Гринвальда (Rick Greenwald) и Дэвида К. Крейнса (David C. Kreines).

«Oracle PL/SQL Language Pocket Reference» (Карманный справочник по языку Oracle PL/SQL), авторы Стивен Фейерштейн (Steven Feuerstein), Билл Прибыл (Bill Pribyl) и Чип Дэйвс (Chip Dawes)

Небольшой, но очень полезный краткий справочник, легко помещающийся в карман. Описывает синтаксис языка PL/SQL вплоть до версии Oracle Database 10*g*.

«Oracle PL/SQL Built-ins Pocket Reference» (Карманный справочник по встроенным пакетам и функциям Oracle PL/SQL), авторы Стивен Фейерштейн (Steven Feuerstein), Джон Бересниевич (John Beresniewicz) и Чип Дэйвс (Chip Dawes)

Еще одно полезное и лаконичное руководство по встроенным функциям и пакетам для Oracle8.

Компакт-диск «Oracle PL/SQL CD Bookshelf»

Предлагает электронные версии большинства из перечисленных выше книг. Актуален для СУБД Oracle8*i*.

PL/SQL в Интернете

Также существует несколько замечательных сетевых ресурсов, которые помогут вам усовершенствовать свои знания по PL/ SQL.

Oracle Technology Network

Присоединяйтесь к сети Oracle Technology Network (OTN), которая «предлагает услуги и ресурсы, необходимые разработчикам для создания, тестирования и развертывания приложений» на основе технологии Oracle. Собравшая в свои ряды миллионы членов, сеть OTN – замечательное место, откуда можно скачать программное обеспечение Oracle, документацию и массу примеров кода. *http://otn.oracle.com*.

---

[1] Рик Гринвальд и Дэвид Крейнс «Oracle. Справочник». – Пер. с англ. – СПб.: Символ-Плюс, 2005.

Quest Pipelines

Quest Software предлагает присоединиться к «свободному интернет-сообществу, созданному для информирования, обучения и поощрения профессионалов в области IT во всем мире». Портал Quest Pipelines (первоначально называвшийся «PL/SQL Pipeline») предлагает дискуссионные форумы, ежемесячные подборки советов, ресурсы для скачивания и, самое главное, бесплатные консультации для разработчиков и администраторов баз данных всего мира для различных СУБД, включая Oracle, DB2, SQL Server и MySQL. *http://www.quest-pipelines.com*.

PLNet.org

PLNet.org – это хранилище программ с открытым кодом, написанных на PL/SQL и могущих быть полезными для разработчиков на PL/SQL, которое поддерживается Биллом Прибылом. Вы можете узнать больше из описания проекта или из ответов на часто задаваемые вопросы (FAQ). Вам предложат ряд полезных программ, например utPLSQL, используемую для автоматизированного тестирования модулей PL/SQL. *http:// plnet.org*.

Open Directory Project

Благодаря проекту «dmoz» (Directory Mozilla) здесь находится коллекция ссылок на сайты, посвященные PL/SQL. Имеется также подкаталог «Tools» (Инструменты) с большим набором ссылок на коммерческие и некоммерческие программы для разработчиков. *http://dmoz.org/Computers/Programming/Languages/PL-SQL/*.

Сайт Стивена Фейерштейна Oracle PL/SQL Programming

На этом сайте предлагаются обучающие курсы, программы для скачивания и другие ресурсы для программистов на PL/SQL, разработанные главным образом Стивеном Фейерштейном. Вы можете скачать материалы всех его семинаров с приложенным кодом. Примеры из этой книги также находятся там. *http://www.oracleplsqlprogramming.com*.

utPLSQL

utPLSQL – это программа с открытым кодом, предназначенная для тестирования модулей PL/SQL. Вы можете использовать ее для стандартизации и автоматизации процесса тестирования. *http://utplsql.sourceforge.net*.

Qnxo

Qnxo (Quality In, Excellence Out) – это разработанный Стивеном Фейерштейном продукт для активного управления процессом разработки, помогающий более эффективно создавать, повторно использовать и тестировать код. В него входит репозиторий, содержащий сотни шаблонов и программ для повторного использования. *http://www.qnxo.com*.

О коде

Все фрагменты кода, использованные в книге, представлены на веб-сайте книги, попасть на который можно с сайта O'Reilly:

http://www.oreilly.com/catalog/oracleplsqldba

и выберите ссылку «Examples» (Примеры).

Мы также рекомендуем посетить «PL/SQL-портал» Стивена Фейерштейна по адресу:

http://www.oracleplsqlprogramming.com

где вы сможете найти обучающие материалы, примеры кода для скачивания и многое другое. На портале также доступны все примеры из нашей книги.

Для того чтобы найти на веб-сайте книги какой-то конкретный фрагмент кода, используйте имя файла, приведенное в тексте. В большинстве случаев имена файлов приводятся в начале соответствующих примеров в виде комментариев:

```
/* File on web: fullname.pkg */
```

Использование примеров кода

Цель этой книги заключается в том, чтобы помочь вам в вашей работе. В общем и целом допускается использование примеров кода данной книги в своих программах и документах. Запрашивать разрешение у компании O'Reilly следует лишь в том случае, когда вы воспроизводите у себя значительный объем кода. То есть создание программы, использующей несколько фрагментов кода, приведенных в данной книге, не требует получения каких-то разрешений. Продажа или распространение компакт-дисков с примерами из книг издательства O'Reilly *требует* получения соответствующего разрешения. Ответ на вопрос с помощью цитаты из нашей книги, как и цитирование фрагмента кода, не требует получения разрешения. Включение значительного объема кода из данной книги в вашу производственную документацию *требует* получения специального разрешения.

Мы были бы признательны (хотя и не требуем этого) за приведение ссылки на источник информации. Такая ссылка обычно включает в себя название, автора, издателя и номер ISBN, например «Oracle PL/SQL for DBAs» by Arup Nanda and Steven Feuerstein. Copyright 2006 O'Reilly Media, Inc., 0-596-00587-3.

Если вам кажется, что ваше использование наших примеров программ выходит за рамки допустимого добросовестного использования, без колебаний обращайтесь к нам по адресу *permissions@oreilly.com*.

Вопросы и замечания

Мы протестировали и проверили данные в этой книге и в исходных текстах настолько хорошо, насколько это возможно, но, учитывая объем информации и быстрое изменение технологии, допускаем, что какие-то функции могли измениться, а мы могли сделать какие-то ошибки. Если вы обнаружите неточности, пожалуйста, сообщите нам об этом по адресу:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в США и Канаде)
707-829-0515 (местный или международный)
707-829-0104 (факс)

Вы также можете отправить сообщение по электронной почте. Для того чтобы попасть в список рассылки или запросить каталог, отправьте электронное письмо по адресу:

info@oreilly.com

Для ответов на технические вопросы и замечаний по книге пишите по адресу:

bookquestions@oreilly.com

В предыдущем разделе мы говорили о том, что у книги есть свой вебсайт, где представлены фрагменты кода, обновленные ссылки и перечень найденных опечаток и ошибок, а также их исправлений. Адрес этого сайта:

http://www.oreilly.com/catalog/oracleplsqldba

Дополнительную информацию об этой и других книгах вы найдете на веб-сайте O'Reilly:

http://www.oreilly.com

Safari® Enabled

Если на обложке технической книги есть пиктограмма «Safari® Enabled», это означает, что книга доступна в Сети через O'Reilly Network Safari Bookshelf.

Safari предлагает намного лучшее решение, чем электронные книги. Это виртуальная библиотека, позволяющая без труда находить тысячи лучших технических книг, вырезать и вставлять примеры кода, загружать главы и находить быстрые ответы, когда требуется наиболее верная и свежая информация. Она свободно доступна по адресу *http://safari.oreilly.com*.

Благодарности

В первую очередь мы хотели бы поблагодарить Дерила Херли (Darryl Hurley), который написал две главы: «Курсоры» и «Табличные функции». Он вступил в дело в решающий момент, принял на себя значительные обязательства и с честью их выполнил. Благодаря ему книга стала значительно лучше. Брин Левеллин (Bryn Llewellyn), менеджер продукта PL/SQL в Oracle, предоставил важнейшую информацию о новых возможностях Oracle Database 10*g* и ответил на множество наших вопросов по различным аспектам PL/SQL.

Нам очень помогли наши технические редакторы: кроме всего прочего мы просили их проверить все фрагменты кода и программы в книге, чтобы свести к минимуму количество ошибок в печатной версии. Мы чрезвычайно благодарны всем специалистам по Oracle PL/SQL, которые потратили часть своего драгоценного времени на то, чтобы книга «Oracle PL/SQL для администраторов баз данных» была как можно лучше. Джеффри Хантер (Jeffrey Hunter) в условиях жесткого цейтнота тщательно проверил все четыре главы, посвященные безопасности, и мы бесконечно благодарны ему за это. Дэниэл Вонг (Daniel Wong) также оказал неоценимый вклад в создание глав по безопасности. Наши искренние благодарности редакторам других глав: Джону Бересниевичу (John Beresniewicz), Дуэйну Кингу (Dwayne King), Стиву Джексону (Steve Jackson), Лорейн Поклингтон (Lorraine Pocklington), Махраж Мадала (Mahraj Madala), Шону О'Кифу (Sean O'Keefe) и Юн-Хо Сикора (Yun-Ho Sikora).

Когда техническая часть была готова, дело перешло в руки замечательной команды O'Reilly Media, возглавляемой нашим добрым другом Деборой Рассел. Они превратили набор глав и примеров кода в книгу, достойную издания в O'Reilly. Огромное спасибо Дарену Келли, руководившему выпуском нашей книги, Робу Романо, создавшему замечательные рисунки, и всей остальной команде.

Аруп благодарен жене Аниндите и сыну Анишу, пожертвовавшим временем, которое семья могла бы провести вместе, ради того, чтобы эта книга появилась на свет. Особое спасибо Анишу, который был слишком мал для того, чтобы выразить свое недовольство словами, хотя, очевидно, был ужасно расстроен тем, что папа не играет с ним.

Стивен благодарит жену Веву Сильва и сыновей Криса Сильва и Эли Сильва Фейерштейнов за их поддержку и понимание того, почему он уделил этой книге столько своего времени и внимания.

1

Введение в PL/SQL

PL/SQL – это процедурное расширение языка SQL (Structured Query Language – структурированный язык запросов). SQL сегодня является повсеместно распространенным языком для выполнения запросов *и* изменений (хоть в его названии об этом и не говорится) в реляционных базах данных. Корпорация Oracle ввела в употребление PL/SQL для того, чтобы избавиться от некоторых ограничений, существующих в SQL, а также для того, чтобы иметь возможность предложить более полное программное решение разработчикам жизненно важных приложений, работающих с базами данных Oracle. В этой главе рассказывается о происхождении языка PL/SQL и приводится краткий обзор основных его элементов.

Мы не надеемся на то, что, прочитав эту главу, вы сразу же сможете писать блестящие программы на PL/SQL. Мы лишь хотим быть уверены в том, что ваших знаний о языке окажется достаточно для понимания и работы с примерами и описаниями функциональности, приводимыми далее в книге. Уделяя особое внимание тем особенностям языка, которые наиболее интересны для администратора баз данных, мы также хотели показать вам всю широту и мощь PL/SQL.

Конечно, в эту главу невозможно вместить все сведения о PL/SQL. Если вы никогда ранее не писали программ и сценариев на PL/SQL, то, вероятно, вам стоит обратиться за дополнительной информацией к двум книгам издательства O'Reilly: «Learning Oracle PL/SQL» (Изучаем Oracle PL/SQL) и «Oracle PL/SQL Programming» (Программирование на Oracle PL/SQL).

Что такое PL/SQL?

Язык Oracle PL/SQL имеет ряд определяющих характеристик:

PL/SQL – это высокоструктурированный, удобочитаемый и доступный язык.

PL/SQL отлично подходит для начинающих программистов. Язык несложен в изучении, названия его многочисленных ключевых слов и структур явно указывают на то, что делает данный фрагмент кода. Если вы знакомы с другими языками программирования, то без труда привыкнете к новому синтаксису.

PL/SQL – это стандартный и переносимый язык для разработки приложений на Oracle.

Написав на своем компьютере PL/SQL-процедуру или функцию для работы с базой данных Oracle, вы можете затем перенести эту процедуру в базу данных своей корпоративной сети и выполнять ее без каких бы то ни было изменений (естественно, при условии совместимости версий Oracle). Принцип «Write once, run everywhere» (написав однажды, запускай везде) был девизом PL/SQL задолго до появления Java. Для PL/SQL «везде» понимается как «везде, где есть база данных Oracle».

PL/SQL – это встроенный язык.

PL/SQL создавался не для автономной работы, а для того чтобы выполняться в определенной среде. Например, вы можете запускать программы на PL/SQL в базе данных (скажем, через интерфейс SQL*Plus). Вы также можете создать программу на PL/SQL и вызвать ее из формы или отчета Oracle Developer (так называемый «клиентский PL/SQL»). Однако невозможно создать исполняемый файл PL/SQL, который выполнялся бы сам по себе.

PL/SQL – это высокопроизводительный и высокоинтегрированный язык для работы с базами данных.

В наше время существует широкий выбор средств, которые можно применять при создании приложений, работающих с базами данных Oracle. Можно использовать Java и JDBC, Visual Basic и ODBC, Delphi, C++ и т. д. Однако вы увидите, что проще всего написать эффективный код для доступа к базе данных Oracle именно на PL/SQL. В частности, Oracle предлагает некоторые специальные дополнительные возможности для PL/SQL, такие как конструкция `FORALL`, которые могут на порядок повысить производительность базы данных.

Основные элементы синтаксиса PL/SQL

В этом разделе вы познакомитесь с основами организации и синтаксиса программы на PL/SQL: структурой блока, набором символов, а также правилами для идентификаторов, разделителей операторов и комментариев.

Структура блока PL/SQL

Как и в большинстве процедурных языков, в PL/SQL наименьшей значимой единицей группировки кода является *блок*. Блок – это конструкция, обеспечивающая выполнение фрагмента кода и определяющая границы видимости переменных и область действия обработчиков исключений. PL/SQL позволяет создавать *анонимные блоки* (блоки кода, не имеющие названия) и *именованные блоки* (это могут быть процедуры, функции или триггеры).

В последующих разделах мы рассмотрим структуру блока и подробно остановимся на анонимных блоках. Различные виды именованных блоков будут описаны далее в главе.

Разделы блока

Блок PL/SQL может включать в себя до четырех разделов (рис. 1.1), лишь один из которых является обязательным.

Заголовок

Используется только для именованных блоков. Заголовок определяет, каким образом будет вызываться именованный блок или программа. Необязательный раздел.

Раздел объявлений

Определяет переменные, курсоры и подблоки, которые упоминаются в разделах исполнения и исключений. Необязательный раздел.

Раздел исполнения

Содержит операторы, которые будет выполнять ядро PL/SQL при исполнении блока. Обязательный раздел.

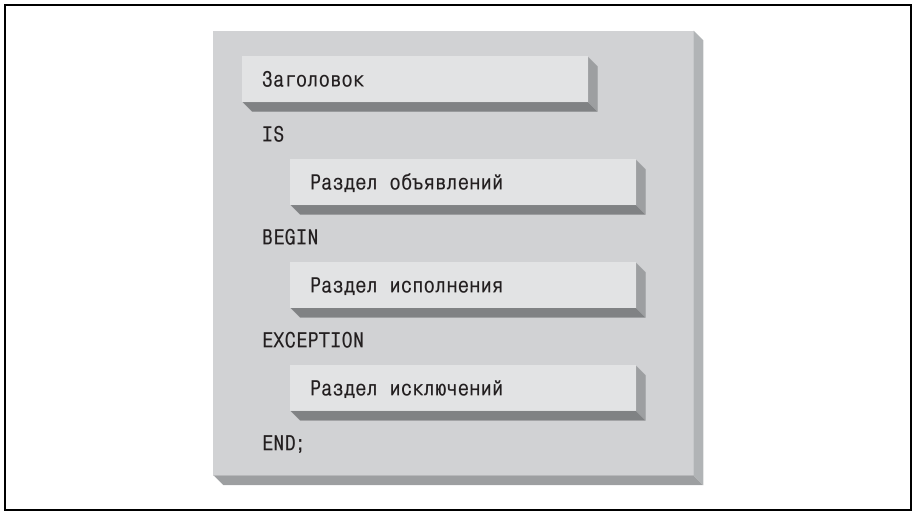


***Рис. 1.1.** Структура блока PL/SQL*

Раздел исключений

Обрабатывает исключительные (по отношению к нормальной работе) ситуации (предупреждения и ошибки). Необязательный раздел.

Анонимные блоки

Если кто-то хочет сохранить анонимность, он не называет своего имени. Именно так и поступает анонимный блок в PL/SQL (рис. 1.2): в нем просто отсутствует раздел заголовка, такой блок начинается с DECLARE или BEGIN. Это означает, что его нельзя будет вызвать из какого-то другого блока, так как не на что установить ссылку. Анонимные блоки служат контейнерами для операторов PL/SQL и обычно включают в себя вызовы процедур и функций.

В общем виде синтаксис анонимного блока PL/SQL будет таким:

```
[ DECLARE
   ... объявления ... ]
BEGIN
   ... один или несколько исполняемых операторов ...
[ EXCEPTION
   ... операторы обработки исключений ... ]
END;
```

В квадратные скобки заключены необязательные элементы конструкции. В блоке должны быть операторы BEGIN и END, а также хотя бы один исполняемый оператор. Рассмотрим несколько примеров анонимных блоков:

- Наиболее короткий анонимный блок:

```
BEGIN
   DBMS_OUTPUT.PUT_LINE(SYSDATE);
END;
```

- Блок с такой же функциональностью, в который добавлен раздел объявлений:

```
DECLARE
   l_right_now VARCHAR2(9);
BEGIN
   l_right_now := SYSDATE;
   DBMS_OUTPUT.PUT_LINE(l_right_now);
END;
```

```
BEGIN                                          ← Только исполнение
   DBMS_OUTPUT.PUT_LINE ('Hello world');
END;
```

***Рис. 1.2.** Анонимный блок, не имеющий разделов объявлений и исключений*

- Тот же блок с добавленным обработчиком исключений:

```
DECLARE
   l_right_now VARCHAR2(9);
BEGIN
   l_right_now := SYSDATE;
   DBMS_OUTPUT.PUT_LINE(l_right_now);
EXCEPTION
   WHEN VALUE_ERROR
   THEN
      DBMS_OUTPUT.PUT_LINE('I bet l_right_now is too small '
         || 'for the default date format!')
END;
```

Набор символов PL/SQL

Программа на PL/SQL состоит из последовательности операторов, каждый из которых образован одной или несколькими строками текста. Набор символов, из которых можно составлять эти строки текста, зависит от используемого в базе данных набора символов. Например, в таблице 1.1 приведен перечень символов, доступных в наборе US7ASCII.

Таблица 1.1. Символы из набора US7ASCII, доступные в PL/SQL

Тип	Символы
Буквы	A–Z, a–z
Цифры	0–9
Символы	~!@#$%*()_-+=|:;»'<>,.?/^
Пробельные символы	Знак табуляции, знак пробела, перевод каретки, конец строки

Любое ключевое слово, оператор и лексема PL/SQL состоит из различных комбинаций символов данного набора символов. Вам нужно лишь понять, как правильно собирать их вместе!

Помните, что PL/SQL *нечувствителен к регистру*, то есть не имеет значения, как вы набираете ключевые слова и идентификаторы. Заглавные буквы воспринимаются так же, как строчные, если только они не выделены специальными разделителями, превращающими их в строковый литерал. Для удобства восприятия авторы этой книги решили использовать верхний регистр для встроенных ключевых слов языка, а нижний – для идентификаторов, определяемых программистом.

Ряд символов (как по отдельности, так и в сочетаниях с другими символами) имеет специальное значение в PL/SQL (табл. 1.2).

Группы символов образуют *лексемы*, которые также называют *атомарными единицами* языка, так как они являются его наименьшими неделимыми составляющими. Лексемами в PL/SQL являются идентификаторы, литералы, разделители и комментарии. Им посвящены последующие разделы.

Таблица 1.2. Простые и составные специальные символы в PL/SQL

Символ	Описание
;	Точка с запятой завершает объявления и операторы.
%	Знак процента является указателем атрибутов (атрибуты курсора, такие как %ISOPEN и атрибуты косвенного объявления, как %ROWTYPE); также используется как многобайтный групповой символ в условии LIKE.
_	Одиночный символ подчеркивания: одиночный групповой символ в условии LIKE.
@	Знак @ указывает на удаленное местоположение.
:	Двоеточие является указателем хост-переменной, как :block.item в Oracle Forms.
**	Двойная звездочка – это оператор возведения в степень.
<> или != или ^= или ~=	Способы обозначения оператора отношения «не равно».
||	Двойная вертикальная черта – это знак операции конкатенации.
<< и >>	Разделители меток.
<= и >=	Операторы отношений «меньше или равно» и «больше или равно».
:=	Оператор присваивания.
=>	Оператор связывания для связывания по имени.
..	Две точки – оператор диапазона.
--	Двойной дефис служит указателем однострочного комментария.
/* и */	Начальный и конечный ограничители многострочного комментария.

Идентификаторы

Идентификатор – это имя объекта PL/SQL (имя переменной или программы, зарезервированное слово). По умолчанию идентификаторы должны обладать следующими свойствами:

- Иметь длину до 30 символов
- Должны начинаться с буквы
- Могут включать в себя знаки доллара $, подчеркивания (_) и диеза (#)
- Не могут содержать никакие пробельные символы

Если два идентификатора отличаются только регистром одной или нескольких букв, то PL/SQL воспринимает их как один и тот же идентификатор. Например, следующие идентификаторы для PL/SQL идентичны:

```
lots_of_$MONEY$  LOTS_of_$MONEY$  Lots_of_$Money$
```

Значения NULL

Отсутствие значения отображается в Oracle при помощи ключевого слова NULL. В предыдущем разделе было показано[1], что переменные почти всех типов данных PL/SQL могут находиться в неопределенном состоянии (исключением являются ассоциативные массивы, экземпляры которых ни при каких условиях не могут быть неопределенными). Обработка значений NULL любого типа данных может вызывать определенные сложности у программиста, при этом строковые значения заслуживают особого упоминания.

В Oracle SQL и PL/SQL строковое значение NULL *обычно* неотличимо от литерала, состоящего из нулевого количества символов (`''` – две последовательные одинарные кавычки, между которыми нет никаких символов). Например, следующее выражение будет вычислено как TRUE и в SQL, и в PL/SQL:

```
'' IS NULL
```

Значение NULL ведет себя так, как если бы типом данных по умолчанию для него являлся VARCHAR2, но сервер Oracle будет пытаться выполнить его неявное преобразование к типу данных, соответствующему выполняемой операции. В некоторых ситуациях от вас может потребоваться явное приведение типов (с использованием такой синтаксической конструкции, как TO_NUMBER(NULL) или CAST(NULL AS NUMBER)).

Литералы

Литерал – это значение, которое не представлено идентификатором, то есть просто значение, которое существует само по себе.

Строковые литералы

Строковый литерал – это текст, заключенный в одинарные кавычки, например:

```
'What a great language!'
```

В отличие от идентификаторов, строковые литералы в PL/SQL чувствительны к регистру, то есть следующие два литерала воспринимаются как различные:

```
'Steven'
'steven'
```

Так что следующее условие будет вычислено как FALSE:

```
IF 'Steven' = 'steven'
```

---

[1] К сожалению, это ошибка автора. В предыдущем разделе ничего об этом не говорилось. – *Примеч. перев.*

Числовые литералы

Числовые литералы могут быть целыми или вещественными числами (то есть содержать дробную часть). Имейте в виду, что PL/SQL считает число 154.00 вещественным числом типа NUMBER, несмотря на то, что его дробная часть равна нулю и на самом деле число является целым. Целые и вещественные числа имеют разное внутреннее представление, и преобразование из одних в другие влечет за собой некоторые дополнительные накладные расходы.

Для записи числового литерала также можно использовать экспоненциальный формат. Буква «E» (в верхнем или нижнем регистре) в записи числа означает его умножение на 10 в соответствующей степени, например 3.05E19, 12e-5.

Начиная с версии Oracle Database 10*g* Release 1 вещественное число может относиться к типу Oracle NUMBER или к стандартному типу IEEE 754 с плавающей точкой. Литералы с плавающей точкой могут иметь двоичное представление с обычной (32 бита, в конце ставится буква «F») или с двойной точностью (64 бита, в конце ставится буква «D»).

В некоторых выражениях можно использовать именованные константы (табл. 1.3), определенные в стандарте IEEE (Institute of Electrical and Electronics Engineers – Институт инженеров по электротехнике и электронике).

Таблица 1.3. Именованные константы для BINARY_FLOAT и BINARY_DOUBLE

Описание	BINARY_FLOAT (32 бита)	BINARY_DOUBLE (64 бита)
Нечисло («Not a number» – NaN); результат деления на ноль или некорректной операции	BINARY_FLOAT_NAN	BINARY_DOUBLE_NAN
Положительная бесконечность	BINARY_FLOAT_INFINITY	BINARY_DOUBLE_INFINITY
Наибольшее конечное число, не превышающее порог переполнения	BINARY_FLOAT_MAX_NORMAL	BINARY_DOUBLE_MAX_NORMAL
Наименьшее нормальное число; порог потери значимости	BINARY_FLOAT_MIN_NORMAL	BINARY_DOUBLE_MIN_NORMAL
Наибольшее положительное число, не превышающее порог переполнения	BINARY_FLOAT_MAX_SUBNORMAL	BINARY_DOUBLE_MAX_SUBNORMAL
Наименьшее число, которое может быть представлено	BINARY_FLOAT_MIN_SUBNORMAL	BINARY_DOUBLE_MIN_SUBNORMAL

Логические литералы

PL/SQL предлагает два литерала для представления логических (булевых) значений: TRUE и FALSE. Эти значения не являются строковыми, их не следует заключать в кавычки. Используйте логические литералы для присваивания значений логическим переменным, например:

```
DECLARE
   enough_money BOOLEAN; -- Объявление логической переменной
BEGIN
   enough_money := FALSE; -- Присваивание ей значения
END;
```

При проверке значения логического выражения не обязательно ссылаться на литерал. Выражение будет «говорить само за себя», как в условии следующего оператора IF:

```
DECLARE
   enough_money BOOLEAN;
BEGIN
   IF enough_money
   THEN
      ...
```

Логическое выражение, переменная или константа также могут принимать значение NULL, что не есть ни TRUE, ни FALSE.

Разделитель «точка с запятой»

Программа PL/SQL состоит из последовательности объявлений и операторов, границы которых определяются не физически, а логически. Другими словами, они не заканчиваются вместе с физическим концом строки кода, а завершаются символом «точка с запятой» (;). Один оператор для удобства восприятия часто записывается на нескольких строках. Например, следующий оператор IF занимает четыре строки, причем для более явного отображения логики в записи используются отступы:

```
IF salary < min_salary(2003)
THEN
   salary := salary + salary * .25;
END IF;
```

В этом операторе IF вы видите две точки с запятой. Первая точка с запятой указывает на конец отдельного исполняемого оператора внутри конструкции IF-END. Вторая точка с запятой обозначает конец самого оператора IF.

Комментарии

Важной составляющей хорошей программы является встроенная документация – *комментарии*. Даже если вы используете в своих PL/SQL-программах удобную модульную структуру и «говорящие» названия,

этого обычно бывает недостаточно для того, чтобы обеспечить полное понимание сложной программы.

PL/SQL поддерживает два вида комментариев: однострочные и многострочные.

Синтаксис однострочного комментария

Однострочный комментарий начинается двумя дефисами (--), между которыми не может стоять пробел или какой-либо другой символ. Весь текст после двойного дефиса и до физического конца строки воспринимается как комментарий и игнорируется компилятором. Если двойной дефис стоит в начале строки, то вся строка является комментарием.

В следующем операторе IF для пояснения его логики использован однострочный комментарий:

```
IF salary < min_salary (2003) --Функция возвращает минимал.годовую зарплату.
THEN
   salary := salary + salary*.25;
END IF;
```

Синтаксис многострочного комментария

Однострочные комментарии удобны для создания кратких пояснений к фрагментам кода или для временного исключения строки программы из исполнения, тогда как многострочные комментарии позволяют включать в программу длинные поясняющие тексты.

Многострочные комментарии начинаются после символов «косая черта-звездочка» (/*) и заканчиваются символами «звездочка-косая черта» (*/). Весь текст, находящийся между этими двумя последовательностями символов, PL/SQL воспринимает как комментарий, и компилятор его игнорирует.

Рассмотрим в качестве примера многострочного комментария блок текста в заголовке процедуры. Символы вертикальной черты в левой части строк использованы для того, чтобы заострить внимание читателя на комментарии:

```
PROCEDURE calc_revenue (company_id IN NUMBER)
/*
| Программа: calc_revenue
| Автор: Стивен Фейерштейн
*/
IS
```

Программные данные

Практически любой написанный вами блок PL/SQL будет определять *программные данные* и их обрабатывать. Программные данные представляют собой структуры данных, которые существуют только в рам-

ках вашего сеанса (физически они находятся в программной глобальной области (Program Global Area – PGA) вашего сеанса) и не хранятся в базе данных. В этом разделе будет рассказано, как объявлять программные данные и какие правила следует соблюдать при выборе имен. Кроме того, вам будет предложен краткий обзор различных типов данных, поддерживаемых в PL/SQL.

Прежде чем вы сможете приступить к работе с переменной или константой, ее необходимо объявить, а после объявления назначить ей имя и тип данных.

При выборе имен для переменных, констант и типов данных необходимо следовать двум основным рекомендациям:

Обязательно убедитесь в том, что каждое название точно отражает назначение объекта и понятно с первого взгляда

Возможно, стоит даже потратить некоторое время на то, чтобы попытаться осознать (вне компьютерной терминологии), что представляет собой конкретная переменная. Тогда вам легко будет подобрать подходящее имя. Например, если переменная будет хранить «общее количество звонков по поводу остывшего кофе», то удачным выбором имени могло бы быть `total_calls_on_cold_coffee` или `tot_cold_calls`, если вы не выносите слишком длинных названий. Примером неудачного выбора могли бы быть имена `totcoffee` и `t_#_calls_lwcoff`, которые слишком загадочны для того, чтобы что-то прояснить.

Выработайте разумные и последовательные соглашения об именовании

Подобные соглашения обычно касаются применения префиксов и/или суффиксов для отражения типа и назначения. Например, имена всех локальных переменных должны начинаться с префикса «l_», а имена глобальных переменных, определяемых в пакете, должны иметь префикс «g_». Имена типов записей должны включать в себя суффикс «_rt» и т. д. Вы можете скачать полный набор соглашений об именовании с веб-страницы Oracle O'Reilly, расположенной по адресу *http://oracle.oreilly.com* (выберите «Oracle PL/SQL Best Practices», затем «Examples»). После скачивания вы сможете пользоваться данным стандартным документом. (В настоящее время его прямым адресом является *http://examples.oreilly.com/orbestprac/*.)

Типы данных PL/SQL

При объявлении переменной или константы вы должны назначить ей тип данных. (PL/SQL за очень небольшими исключениями является языком со *строгой типизацией*.) PL/SQL предлагает полный набор предопределенных скалярных и составных типов данных, вы также можете создавать собственные пользовательские типы (которые также называют *абстрактными типами данных*).

Все имеющиеся предопределенные типы данных определены в PL/SQL-пакете STANDARD. Например, туда включены операторы, определяющие логический тип данных и два числовых типа:

```
CREATE OR REPLACE PACKAGE STANDARD
IS
  type BOOLEAN is (FALSE, TRUE);
  type NUMBER is NUMBER_BASE;
  subtype INTEGER is NUMBER(38,);
```

PL/SQL поддерживает все привычные типы данных и множество других. В разделе будет приведен лишь краткий обзор разнообразных предопределенных типов данных.

Символьные типы

PL/SQL поддерживает строки как фиксированной, так и переменной длины, представленные как в традиционных кодировках, так и в кодировках Unicode. CHAR и NCHAR – это типы строк фиксированной длины, а VARCHAR2 и NVARCHAR2 – типы строк переменной длины. Рассмотрим объявление строки переменной длины, которая может вмещать до 2000 символов:

```
DECLARE
   l_accident_description VARCHAR2(2000);
```

Oracle также поддерживает очень длинные символьные строки – типы LONG и LOB. Эти типы данных позволяют хранить и обрабатывать огромные объемы данных: LOB может содержать до 128 терабайт информации в Oracle Database 10*g* (используйте тип LONG только для совместимости с уже существующим кодом. Будущее за типами LOB!). К символьным типам данных LOB относятся CLOB (character large object – большой символьный объект) и NCLOB (National Language Support character large object – большой символьный объект с поддержкой национальных языков, многобайтный формат).

Числовые типы

PL/SQL поддерживает все более широкое множество числовых типов данных. Долгие годы рабочей лошадкой числовых типов данных был тип NUMBER, который можно использовать для десятичных значений с фиксированной и плавающей точкой, а также для целых значений. Приведем несколько примеров объявлений типа NUMBER:

```
DECLARE
   salary NUMBER(9,2); -- фиксированная точка, семь знаков слева и два справа
   raise_factor NUMBER; -- десятичное число с плавающей точкой
   weeks_to_pay NUMBER(2); -- целое число
BEGIN
   salary := 1234567.89;
   raise_factor := 0.05;
   weeks_to_pay := 52;
END;
```

Десятичная природа типа NUMBER оказывается чрезвычайно полезной при работе с денежными величинами. Вам не придется беспокоиться о возможных ошибках округления при переводе числа в двоичное представление. Например, записывая число 0.95, не стоит бояться, что от него через некоторое время останется только 0.949999968.

До выпуска версии Oracle Database 10*g* тип NUMBER был единственным числовым типом данных PL/SQL, полностью соответствующим типу данных базы данных. Это одна из причин столь широкого использования типа NUMBER. В Oracle Database 10*g* появилось еще два двоичных типа с плавающей точкой: BINARY_FLOAT и BINARY_DOUBLE. Как и NUMBER, оба новых типа поддерживаются как в PL/SQL, так и в базе данных. Правильно применяя их, можно добиться значительного повышения производительности за счет того, что математические операции над новыми типами выполняются аппаратной частью (когда это позволяет аппаратная платформа).

PL/SQL поддерживает ряд числовых типов и подтипов, которые не соответствуют типам базы данных, но, тем не менее, весьма полезны. Упомянем особо PLS_INTEGER, целочисленный тип, для которого арифметические операции выполняются аппаратно. Счетчики циклов FOR реализованы типом PLS_INTEGER.

Даты, временные метки и интервалы

До появления версии Oracle9*i* Database мир дат Oracle ограничивался типом DATE, который позволял хранить как дату, так и время (с точностью до секунд). В Oracle9*i* Database появились два набора новых связанных типов данных: INTERVAL и TIMESTAMP. Новые типы значительно расширили возможности разработчиков PL/SQL по созданию программ, обрабатывающих и хранящих значения дат и времени с очень высокой точностью, а также вычисляющих и хранящих интервалы времени.

Приведем в качестве примера функцию, вычисляющую возраст человека:

```
CREATE OR REPLACE FUNCTION age (dob_in IN DATE)
   RETURN INTERVAL YEAR TO MONTH
IS
   retval INTERVAL YEAR TO MONTH;
BEGIN
   RETURN (SYSDATE - dob_in) YEAR TO MONTH;
END;
```

Логические типы

PL/SQL поддерживает настоящий логический (булев) тип данных. Переменная этого типа может иметь лишь одно из трех значений: TRUE, FALSE и NULL.

Логические переменные позволяют сделать код удобочитаемым, даже в том случае, когда он содержит сложные логические выражения. Рас-

смотрим пример объявления переменной типа Boolean с присваиванием ей значения по умолчанию:

```
DECLARE
   l_eligible_for_discount BOOLEAN :=
      customer_in.balance > min_balance AND
      customer_in.pref_type = 'MOST FAVORED' AND
      customer_in.disc_eligibility;
```

Двоичные данные

Oracle поддерживает несколько видов *двоичных данных* (это неструктурированные данные, которые не интерпретируются и не обрабатываются Oracle), в том числе RAW, LONG RAW, BFILE и BLOB. Тип данных BFILE хранит неструктурированные двоичные данные в файлах операционной системы вне базы данных. RAW – это тип данных переменной длины, подобный символьному типу данных VARCHAR2 и отличающийся от него тем, что утилиты Oracle не выполняют преобразования символов при передаче данных типа RAW.

ROWID

Oracle поддерживает два собственных типа данных, ROWID и UROWID, которые используются для представления адреса строки в таблице. ROWID – это уникальный адрес строки в соответствующей таблице, а UROWID – логическая позиция строки в индекс-таблице (index-organized table, IOT). ROWID также является SQL-псевдонимом, который может использоваться в командах SQL.

REF CURSOR

Тип данных REF CURSOR позволяет объявлять курсорные переменные, которые могут использоваться со статическими и динамическими командами SQL для реализации чрезвычайно гибких требований. Этот тип данных имеет две разновидности: строгий REF CURSOR и нестрогий REF CURSOR. Нестрогий REF CURSOR – это один из немногих доступных вам типов данных со слабой типизацией.

Рассмотрим пример объявления строгого типа REF CURSOR (ассоциируем курсорную переменную с конкретной записью при помощи атрибута %ROWTYPE):

```
DECLARE
   TYPE  book_data_t IS REF CURSOR RETURN book%ROWTYPE;
   book_curs_var book_data_t;
```

Теперь рассмотрим два объявления нестрогого типа REF CURSOR, в которых никакая конкретная структура не ассоциируется с результирующей переменной. В четвертой строке представлен SYS_REFCURSOR, предопределенный нестрогий тип REF CURSOR.

```
DECLARE
   TYPE  book_data_t IS REF CURSOR;
```

```
book_curs_var book_data_t;
book_curs_var2 SYS_REFCURSOR
```

Типы данных для сети Интернет

В версии Oracle9*i* Database появилась встроенная поддержка различных связанных с Интернетом типов данных и технологий, в частности XML (Extensible Markup Language – расширяемый язык разметки) и URI (Universal Resource Identifiers – универсальные идентификаторы ресурсов). Oracle поддерживает типы данных, используемые для работы с данными XML и URI, а также специальный класс DBUri-REF, который применяется для доступа к данным, хранящимся внутри самой базы данных. Oracle также предоставляет новый набор типов для хранения внешних и внутренних URI и доступа к ним из базы данных.

Тип XMLType позволяет хранить в базе данных данные XML и обращаться к ним с запросами при помощи таких функций, как `SYS_XMLGEN`, и пакета `DBMS_XMLGEN`. Он также позволяет использовать операторы языка SQL для выполнения поиска при помощи языка XPath.

Связанные с URI типы данных, такие как URIType и HttpURIType, входят в иерархию объектных типов. Они могут использоваться для хранения URL-адресов внешних веб-страниц и файлов, а также для ссылок на данные, хранящиеся внутри базы данных.

Типы данных «Any»

Обычно перед программистом стоит вполне конкретная задача с жестко заданными требованиями. Но случается и так, что необходимо написать нечто общее, для широкого применения. В таких случаях удобно использовать типы данных «Any».

Типы «Any» появились в версии Oracle9*i* Database Release 1. Они значительно отличаются от любых других типов данных, доступных в Oracle. Эти типы позволяют динамически инкапсулировать описания типов, экземпляры данных и наборы экземпляров данных любого другого типа SQL, а также обращаться к таким объектам. Вы можете использовать эти типы (и методы, определенные для них как для объектных типов), например для определения типа данных, хранящихся в некоторой вложенной таблице, без обращения к реальному объявлению типа данной таблицы.

Группа типов данных «Any» включает в себя AnyType, AnyData и AnyDataSet.

Объявление программных данных

Как уже говорилось, прежде чем обращаться к переменной или константе, необходимо ее объявить (единственным исключением из этого правила являются индексные переменные циклов `FOR`.) Все объявления должны быть сделаны в разделе объявлений анонимного блока,

процедуры, функции, триггера, тела пакета или тела объектного типа. В PL/SQL вы можете объявить множество типов данных и структур данных, включая переменные, константы, пользовательские типы TYPE (например, для коллекции или записи) и исключения. В этом разделе будет рассказано об объявлении переменных и констант.

Объявление переменных

При объявлении переменной PL/SQL выделяет память для значения переменной и присваивает этому хранилищу имя, используя которое вы сможете извлекать и изменять данное значение. В объявлении также указывается тип данных переменной, который будет использован для проверки корректности значений, присваиваемых переменной.

Для объявления используется следующая синтаксическая конструкция:

```
имя тип_данных [NOT NULL] [значение_по_умолчанию];
```

где *имя* – это имя объявляемой переменной или константы, а *тип_данных* – это тип или подтип значений, которые могут присваиваться этой переменной. Включение в объявление выражения NOT NULL означает, что если в коде будет предпринята попытка присвоения вашей переменной значения NULL, то Oracle инициирует исключение. Выражение [*значение по умолчанию*] позволяет инициализировать переменную; оно необязательно для всех объявлений, кроме объявлений констант.

Рассмотрим пример объявления переменных различных типов данных:

```
DECLARE
   -- Простое объявление числовой переменной
   l_total_count NUMBER;

   -- Объявление числа, округляемого до сотых (центов):
   l_dollar_amount NUMBER (10,2);

   -- Отдельная переменная даты, со значением по умолчанию
   -- «прямо сейчас», которая не может быть NULL
   l_right_now DATE NOT NULL  DEFAULT SYSDATE;

   -- Использование оператора присваивания для указания
   -- значения по умолчанию
   l_favorite_flavor VARCHAR2(100) := 'Anything with chocolate, actually';

   -- Двухэтапное объявление ассоциативного массива.
   -- Сначала табличный тип:
   TYPE list_of_books_t IS TABLE OF book%ROWTYPE INDEX BY BINARY_INTEGER;

   -- Затем тот конкретный список, который будет
   -- обрабатываться в данном блоке:
   oreilly_oracle_books list_of_books_t;
```

Для задания значения по умолчанию ключевое слово DEFAULT и оператор присваивания эквиваленты и взаимозаменяемы. Что же следует

использовать? Я предпочитаю использовать оператор присваивания (:=) для указания значений по умолчанию для констант, а ключевое слово DEFAULT – для переменных. Для констант присваиваемое значение в действительности является не значением по умолчанию, а исходным (и неизменным) значением, поэтому использование DEFAULT кажется мне неуместным.

Объявление констант

Существует всего два отличия в объявлениях переменных и констант: объявление константы включает в себя ключевое слово CONSTANT, и в нем обязательно указывается значение по умолчанию (которое на самом деле является не *значением по умолчанию*, а *единственно возможным* значением). Синтаксис объявления константы будет таким:

```
имя CONSTANT тип_данных [NOT NULL] := | DEFAULT значение_по_умолчанию;
```

Значение константы задается в момент объявления и впоследствии не может быть изменено.

Рассмотрим несколько примеров объявления констант:

```
DECLARE
   -- Текущий год; это значение не изменится в течение сеанса.
   l_curr_year CONSTANT PLS_INTEGER :=
      TO_NUMBER (TO_CHAR (SYSDATE, 'YYYY'));

   -- Использование ключевого слова DEFAULT
   l_author CONSTANT VARCHAR2(100) DEFAULT 'Bill Pribyl';

   -- Объявления объектного типа как константы.
   -- Константы не обязаны быть скалярными!
   l_steven CONSTANT  person_ot :=
     person_ot ('HUMAN', 'Steven Feuerstein', 175, '09-23-1958');
```

Если не указано иное, то информация, приводимая в последующих разделах главы для переменных, справедлива и для констант.

Объявления с привязкой

Привязка типа данных означает, что вы указываете компилятору PL/SQL на необходимость назначения типа данных переменной на основе типа данных уже определенной структуры данных: другой переменной PL/SQL, предопределенного типа или подтипа, таблицы базы данных или какого-то столбца таблицы. PL/SQL поддерживает две разновидности привязки:

Скалярная привязка

Используйте атрибут %TYPE для определения переменной на основе столбца таблицы или какой-то другой скалярной переменной PL/SQL.

Привязка к записи

Используйте атрибут %ROWTYPE для определения записи на основе таблицы или предопределенного явного курсора PL/SQL.

Синтаксис объявления типа данных с привязкой будет таким:

```
имя_переменной атрибут%TYPE [необязательное присваивание
                             значения по умолчанию];
имя_переменной имя_таблицы|имя_курсора%ROWTYPE [необязательное присваивание
                                                значения по умолчанию];
```

где *имя_переменной* – это имя объявляемой переменной, а *атрибут* – это имя объявленной ранее переменной PL/SQL или же указание столбца таблицы в формате «таблица.столбец».

Приведем пример привязки переменной к столбцу таблицы базы данных:

```
l_company_id company.company_id%TYPE;
```

Теперь рассмотрим пример привязки записи к курсору:

```
DECLARE
   CURSOR book_cur IS
      SELECT author, title FROM book;
   l_book book_cur%ROWTYPE;
```

Ссылка на привязку вычисляется при компиляции, что не увеличивает времени выполнения программы. Привязка устанавливает зависимость между кодом и элементом, к которому выполняется привязка (таблицей, курсором или пакетом, содержащим переменную, на которую ссылается объявление типа). При изменении такого элемента привязанный к нему код помечается как INVALID. При повторной компиляции привязка будет восстановлена, так что код всегда будет соответствовать текущему состоянию элемента.

Управляющие операторы

В PL/SQL существует два вида управляющих операторов: условные операторы и операторы перехода. Условные операторы, направляющие поток выполнения в определенную точку программы в зависимости от некоторого условия, необходимы практически в каждом фрагменте создаваемого кода. К таким операторам относятся IF-THEN-ELSE и CASE (операторы CASE доступны в версиях Oracle9*i* Database и Oracle Database 10*g*). Следует отличать операторы CASE от выражений CASE. Выражение CASE в некоторых случаях вполне может заменить собой операторы IF или CASE. Существенно реже используется оператор безусловного перехода GOTO или явное указание на необходимость «ничего-не-делать» с помощью оператора NULL.

Операторы IF

Оператор IF позволяет использовать в программах условную логику. Операторы IF бывают трех видов (табл. 1.4).

Таблица 1.4. Типы операторов IF

Тип оператора IF	Описание
IF *условие* THEN END IF;	Простейшая форма оператора IF. Условие, указанное между IF и THEN, определяет, должно ли быть выполнено множество операторов, находящееся между THEN и END IF. Если условие вычислено как FALSE, то код не выполняется.

IF *условие* THEN
ELSE
END IF;
	Данная конструкция реализует логику «или-или». Вычисляется условие, указанное между IF и THEN, и выполняется фрагмент кода, расположенный между THEN и ELSE, или фрагмент кода, расположенный между ELSE и END IF. Всегда выполняется только один из фрагментов кода.

IF *условие1* THEN
ELSIF *условие2* THEN
ELSE
END IF;
	Последняя и наиболее сложная форма оператора IF. Действие выбирается на основе оценки ряда взаимно исключающих условий, и выполняется соответствующее множество исполняемых операторов. Если вы пишете подобные операторы в версии Oracle9*i* Database Release 1 и выше, то подумайте о том, чтобы использовать вместо них *поисковые* операторы выбора CASE.

Операторы и выражения CASE

Оператор CASE позволяет выбрать для исполнения одну из нескольких последовательностей операторов. Операторы CASE появились в стандарте SQL уже в 1992 году, но Oracle SQL стал поддерживать CASE только в версии Oracle8*i* Database, а PL/SQL не поддерживал CASE вплоть до версии Oracle9*i* Database Release 1. Начиная с этой версии PL/SQL поддерживает следующие виды операторов CASE:

Простой оператор CASE

Ставит в соответствие каждой последовательности операторов PL/SQL некоторое значение. Выбирает последовательность операторов для выполнения на основе вычисления выражения, возвращающего одно из таких значений.

Поисковый оператор CASE

Выбирает последовательность операторов для выполнения на основе вычисления списка логических условий. Выполняется последовательность операторов, соответствующая первому условию, вычисленному как TRUE.

В дополнение к операторам CASE PL/SQL поддерживает также выражения CASE. По форме выражение CASE очень похоже на оператор CASE. Оно позволяет выбрать из множества выражений то выражение, которое должно быть вычислено. Результатом выражения CASE является неко-

торое значение, в то время как результатом оператора CASE является исполнение последовательности операторов PL/SQL.

Простой оператор CASE

Простой оператор CASE позволяет на основе вычисления результатов некоторого выражения выбрать одну из нескольких последовательностей операторов PL/SQL для исполнения. Приведем пример простого оператора CASE, в котором для выбора нужного алгоритма вычисления бонуса используется анализ должности сотрудников (переменная employee type):

```
CASE employee_type
WHEN 'S' THEN
   award_salary_bonus(employee_id);
WHEN 'H' THEN
   award_hourly_bonus(employee_id);
WHEN 'C' THEN
   award_commissioned_bonus(employee_id);
ELSE
   RAISE invalid_employee_type;
END CASE;
```

Рассмотренный оператор CASE содержит явное выражение ELSE, однако в общем случае оно не является обязательным. Если выражение ELSE явно не указано, то PL/SQL неявно использует такую конструкцию:

```
ELSE
   RAISE CASE_NOT_FOUND;
```

Поисковый оператор CASE

Поисковый оператор CASE вычисляет список логических выражений и, найдя выражение, равное TRUE, выполняет соответствующую ему последовательность операторов. Фактически поисковый оператор CASE эквивалентен оператору CASE TRUE из предыдущего раздела. Приведем пример поискового оператора CASE:

```
CASE
WHEN salary >= 10000 AND salary <=20000 THEN
   give_bonus(employee_id, 1500);
WHEN salary > 20000 AND salary <= 40000 THEN
   give_bonus(employee_id, 1000);
WHEN salary > 40000 THEN
   give_bonus(employee_id, 500);
ELSE
   give_bonus(employee_id, 0);
END CASE;
```

Циклы в PL/SQL

PL/SQL поддерживает три вида циклов, обеспечивая тем самым возможность создания оптимального кода для решения любой конкрет-

ной задачи. В большинстве случаев, требующих использования цикла, применима любая из циклических конструкций. Однако выбор не самой удачной для конкретного случая конструкции может повлечь за собой написание множества дополнительных строк кода. В итоге получившийся модуль будет сложнее для восприятия и дальнейшего сопровождения.

Чтобы показать, как разные виды циклов по-разному решают поставленную перед ними задачу, рассмотрим далее три типа циклов. В каждом случае процедура вызывает `display_total_sales` для каждого года, номер которого находится в диапазоне между начальным и конечным значением аргумента.

Очевидно, в рассмотренных примерах цикл `FOR` требует наименьшего объема кода. Но использование данного типа цикла возможно лишь потому, что заранее известно, что тело цикла будет выполняться определенное количество раз. Во множестве других случаев количество проходов цикла должно быть переменным, так что применение цикла `FOR` будет невозможно.

Простой цикл

Простой цикл называется простым, потому что он начинается просто со слова `LOOP` и заканчивается оператором `END LOOP`. Цикл завершается при выполнении внутри цикла оператора `EXIT`, `EXIT WHEN` или `RETURN` (или если внутри цикла инициировано исключение):

```
PROCEDURE display_multiple_years (
   start_year_in IN PLS_INTEGER
  ,end_year_in IN PLS_INTEGER
)
IS
   l_current_year PLS_INTEGER := start_year_in;
BEGIN
   LOOP
      EXIT WHEN l_current_year > end_year_in;
      display_total_sales (l_current_year);
      l_current_year :=  l_current_year + 1;
   END LOOP;
END display_multiple_years;
```

Цикл FOR

Oracle поддерживает циклы `FOR` со счетчиком и с курсором. Для цикла `FOR` со счетчиком вы указываете начальное и конечное целые значения, а все остальное за вас делает PL/SQL: проходит все значения внутри заданного диапазона и завершает цикл:

```
PROCEDURE display_multiple_years (
   start_year_in IN PLS_INTEGER
  ,end_year_in IN PLS_INTEGER
```

```
)
IS
BEGIN
   FOR l_current_year IN start_year_in .. end_year_in
   LOOP
      display_total_sales (l_current_year);
   END LOOP;
END display_multiple_years;
```

Цикл FOR с курсором имеет такую же базовую структуру, только в данном случае вместо указания верхней и нижней границ целочисленного диапазона следует явно задать курсор или использовать оператор SELECT:

```
PROCEDURE display_multiple_years (
   start_year_in IN PLS_INTEGER
  ,end_year_in IN PLS_INTEGER
)
IS
BEGIN
   FOR l_current_year IN (
      SELECT * FROM sales_data
       WHERE year BETWEEN start_year_in AND end_year_in)
   LOOP
      -- Теперь эта процедура принимает запись, неявно
      -- объявленную как sales_data%ROWTYPE...
      display_total_sales (l_current_year);
   END LOOP;
END display_multiple_years;
```

Цикл WHILE

Цикл WHILE очень похож на простой цикл. Его ключевым отличием является то, что цикл WHILE проверяет условие завершения до выполнения тела цикла. Следовательно, тело цикла может не быть выполнено ни разу:

```
PROCEDURE display_multiple_years (
   start_year_in IN PLS_INTEGER
  ,end_year_in IN PLS_INTEGER
)
IS
   l_current_year PLS_INTEGER := start_year_in;
BEGIN
   WHILE (l_current_year <= end_year_in)
   LOOP
      display_total_sales (l_current_year);
      l_current_year :=  l_current_year + 1;
   END LOOP;
END display_multiple_years;
```

Обработка исключений

В языке PL/SQL ошибки любого рода трактуются как *исключения* – нештатные ситуации для вашей программы. Исключения могут быть следующих видов:

- Ошибка, инициированная системой (например, «недостаточно памяти» или «повторение значений в индексе»).
- Ошибка, вызванная действиями пользователя.
- Предупреждение, выдаваемое пользователю приложением.

PL/SQL перехватывает ошибки и реагирует на них, используя механизм обработчиков исключений. Обработчики исключений позволяют аккуратно отделить код обработки ошибок от исполняемых операторов. Для обработки ошибок используется событийная модель исполнения кода, а не линейная. Другими словами, вне зависимости от того, где было инициировано исключение, оно будет обработано одним и тем же обработчиком исключений в разделе исключений.

При возникновении ошибки в PL/SQL, будь то системная ошибка или ошибка приложения, инициируется исключение. Обработка в исполняемом разделе текущего PL/SQL-блока прекращается, и управление передается в отдельный раздел исключений текущего блока (если такой существует) для обработки исключения. После завершения обработки исключения вернуться в этот блок невозможно. Управление передается в родительский блок (если он существует).

Определение исключений

Для того чтобы исключение могло быть инициировано или обработано, оно должно быть определено. В Oracle предопределены тысячи исключений, при этом большинству из них сопоставлены номера и сообщения. Небольшой части этих тысяч исключений присвоены имена – речь идет о самых часто встречающихся исключениях.

Имена исключениям присваиваются в пакете `STANDARD` (один из двух встроенных по умолчанию пакетов PL/SQL), а также в других встроенных пакетах, таких как `UTL_FILE` и `DBMS_SQL`. Для определения исключений, таких как `NO_DATA_FOUND`, Oracle использует точно такой же код, который вы будете использовать для определения или объявления собственных исключений. Определять собственные исключения вы можете двумя разными способами, которые будут описаны в последующих разделах.

Вы можете объявить собственное исключение, указав в разделе объявлений имя исключения, которое вы хотите инициировать в программе, а затем ключевое слово `EXCEPTION`:

```
DECLARE
   имя_исключения EXCEPTION;
```

Имена исключений имеют такой же формат, что и имена переменных[1], но ссылаться на них можно только двумя способами:

- В операторе RAISE в разделе исполнения программы (для инициирования исключения), например:

```
RAISE invalid_company_id;
```

- В предложениях WHEN в разделе исключений (для обработки инициированного исключения), например:

```
WHEN invalid_company_id THEN
```

Инициирование исключений

Исключение в вашем приложении может быть инициировано тремя способами:

- Oracle может инициировать сообщение, обнаружив ошибку.
- Вы можете инициировать исключение при помощи оператора RAISE.
- Вы можете инициировать исключение при помощи встроенной процедуры RAISE_APPLICATION_ERROR.

Мы уже знаем, как инициирует исключения Oracle. Теперь рассмотрим различные механизмы инициирования исключений программистом.

Оператор RAISE

Поддерживаемый Oracle оператор RAISE позволяет программисту инициировать именованные исключения. Вы можете инициировать как системное, так и собственное исключение. Оператор RAISE может иметь одну из трех форм:

```
RAISE имя_исключения;
RAISE имя_пакета.имя_исключения;
RAISE;
```

Первая форма (без указания имени пакета) может использоваться для инициирования исключения, определенного в текущем блоке (или во внешнем блоке, содержащем текущий), или для инициирования системного исключения, определенного в пакете STANDARD.

Вторая форма требует указания имени пакета. Если исключение было объявлено внутри пакета (не пакета STANDARD), а инициируете вы его вне этого пакета, то следует указать ссылку на пакет в операторе RAISE.

Третья форма оператора RAISE не требует указания имени исключения, но может использоваться только внутри предложения WHEN в разделе исключений. Используйте эту форму для повторного инициирования

[1] В оригинале – имена логических переменных. Мы полагаем, что имена формируются аналогично именам любых, не только логических, переменных. – *Примеч. науч. ред.*

исключения из обработчика исключений (другими словами, для передачи исключения дальше, во внешний блок).

Процедура RAISE_APPLICATION_ERROR

В Oracle имеется процедура RAISE_APPLICATION_ERROR (она определена в используемом по умолчанию пакете DBMS_STANDARD) для инициирования ошибок, специфичных для конкретного приложения, информация о которых должна передаваться в среду исполнения программы.

Заголовок процедуры в пакете DBMS_STANDARD выглядит следующим образом:

```
PROCEDURE RAISE_APPLICATION_ERROR (
   num binary_integer,
   msg varchar2,
   keeperrorstack boolean default FALSE);
```

где *num* – это номер ошибки в диапазоне между -20999 и -20000 (только представьте: все остальные отрицательные целые числа Oracle использует для собственных исключений!); *msg* – это сообщение об ошибке, длина которого не должна превышать 2 Кб (любой текст, выходящий за рамки этого ограничения, будет проигнорирован); а *keeperrorstack* указывает на то, хотите ли вы добавить ошибку к уже содержащимся в стеке ошибкам (TRUE) или же заменить ею уже имеющиеся ошибки (значение по умолчанию – FALSE).

Обработка исключений

Как только инициировано исключение, нормальное исполнение текущего PL/SQL-блока прекращается, и управление передается в раздел исключений. Исключение обрабатывается обработчиком исключений текущего PL/SQL-блока или передается для обработки в родительский блок.

Для того чтобы обеспечить перехват и обработку исключений, необходимо написать соответствующий обработчик исключений. Обработчики исключений должны размещаться после всех исполняемых операторов вашей программы, но до оператора END блока. Ключевое слово EXCEPTION обозначает начало раздела исключений и отдельных обработчиков исключений. Синтаксическая конструкция обработчика исключений выглядит так:

```
WHEN имя_исключения [ OR имя_исключения ... ]
THEN
   исполняемые операторы
```

или так:

```
WHEN OTHERS
THEN
   исполняемые операторы
```

Предложение WHEN OTHERS является необязательным. Если оно отсутствует, то все необработанные исключения сразу же передаются в родительский блок (если он имеется). Предложение WHEN OTHERS должно быть последним обработчиком исключений в разделе исключений.

Встроенные функции обработки ошибок

Oracle поддерживает ряд встроенных функций, которые призваны помочь вам выявить, проанализировать и отреагировать на ошибки вашего PL/SQL-приложения.

SQLCODE

Функция SQLCODE возвращает код ошибки для последнего (текущего) исключения в блоке. При отсутствии ошибок SQLCODE возвращает 0. Функция SQLCODE также возвращает 0 в случае, если она вызывается извне обработчика исключений.

SQLERRM

Функция SQLERRM возвращает сообщение об ошибке по ее коду. Если не передать SQLERRM код ошибки, то будет выдано сообщение об ошибке с кодом, возвращенным функцией SQLCODE. Максимальная длина строки, возвращаемой SQLERRM, составляет 512 байт (в некоторых более ранних версиях Oracle – всего 255 байт).

DBMS_UTILITY.FORMAT_ERROR_STACK

Эта встроенная функция, как и SQLERRM, возвращает полное сообщение, соответствующее текущей ошибке (то есть значению, возвращенному функцией SQLCODE). Как правило, следует вызывать эту функцию внутри обработчика ошибок для того, чтобы получить полное сообщение об ошибке.

DBMS_UTILITY.FORMAT_ERROR_BACKTRACE

Эта функция, появившаяся в версии Oracle Database 10*g* Release 1, возвращает форматированную строку, которая отображает стек программ и номера строк, ведущие к месту возникновения ошибки.

Необработанные исключения

Если инициированное в программе исключение не обработано обработчиком исключений ни в текущем, ни в одном из родительских блоков, то это *необработанное* исключение. PL/SQL возвращает ошибку, породившую необработанное исключение, обратно в среду приложения, откуда был запущен PL/SQL. Затем соответствующие действия предпринимаются уже этой средой (программой SQL*Plus, Oracle Forms или Java). В случае с SQL*Plus автоматически выполняется откат (ROLLBACK) всех DML-изменений, выполненных в блоке верхнего уровня.

Передача необработанного исключения

Правила для области действия исключений определяют блок, в котором исключение может быть инициировано. Правила передачи исключений определяют способ их обработки после инициирования.

Когда порождается исключение, PL/SQL ищет обработчик исключений в текущем блоке (анонимном блоке, процедуре или функции). Если обработчик не найден, то PL/SQL передает исключение в родительский блок текущего блока. Затем PL/SQL пытается обработать исключение, инициировав его еще раз в родительском блоке. Процесс продолжается до тех пор, пока не закончатся все последовательные родительские блоки, в которых можно было бы инициировать исключение (рис. 1.3).

Когда все блоки исчерпаны, PL/SQL возвращает необработанное исключение в среду приложения, которое исполняло самый внешний блок PL/SQL. Необработанное исключение прекращает исполнение вызывающей программы.

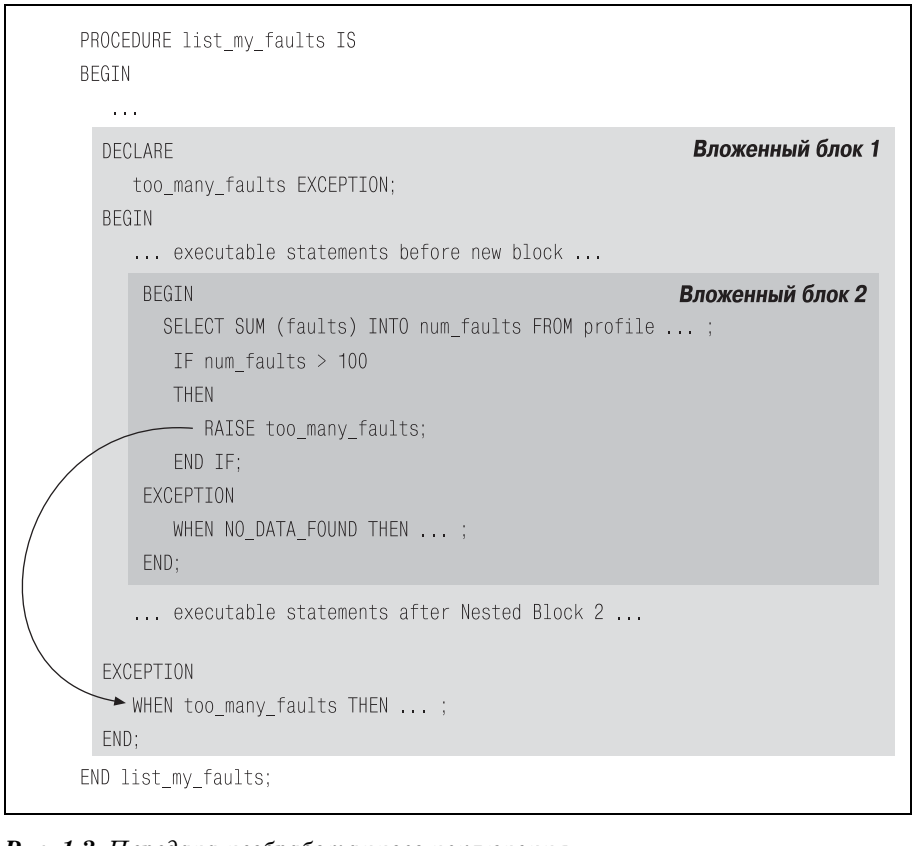


Рис. 1.3. *Передача необработанного исключения*

Записи

Каждая строка таблицы состоит из одного или нескольких столбцов различных типов. Аналогично запись состоит из одного или нескольких полей. Существует три способа определения записи, но после того как она определена, обращение к полям записи и их изменение всегда подчиняются одним и тем же правилам.

Рассмотрим пример объявления записи непосредственно на основе таблицы базы данных. Пусть у меня есть таблица для хранения сведений о моих любимых книгах:

```
CREATE TABLE books (
  book_id         INTEGER,
  isbn            VARCHAR2(13)
  title           VARCHAR2(200),
);
```

Я могу без труда создать запись на основе этой таблицы, заполнить ее результатами запроса к базе данных и затем обращаться к отдельным столбцам как к полям записи:

```
DECLARE
   my_book   books%ROWTYPE;
BEGIN
   SELECT *
     INTO my_book
     FROM books
    WHERE title = 'Oracle PL/SQL Programming, 4th Edition';
END;
```

Объявление записей

Существуют три способа объявления записей:

Запись на основе таблицы

Используйте атрибут `%ROWTYPE` и имя таблицы для объявления записи, в которой каждое поле соответствует столбцу таблицы (и имеет такое же название). Объявим запись `one_book`, которая будет иметь такую же структуру, как и таблица `books`:

```
DECLARE
   one_book books%ROWTYPE;
```

Запись на основе курсора

Используйте атрибут `%ROWTYPE` и явный курсор или курсорную переменную, в которых каждое поле соответствует столбцу или именованному выражению (aliased expression) в операторе `SELECT` курсора. Объявим запись, имеющую такую же структуру, как явный курсор:

```
DECLARE
   CURSOR my_books_cur IS
```

```
      SELECT * FROM books
       WHERE author LIKE '%FEUERSTEIN%';

  one_SF_book my_books_cur%ROWTYPE;
```

Запись, определяемая программистом

Используйте оператор TYPE … RECORD для определения записи, в которой каждое поле явно определено (с указанием имени и типа данных) в операторе TYPE. При этом полем записи, определяемой программистом, может являться также другая запись. В следующем примере я объявляю тип записи TYPE, содержащий некоторую информацию о моей писательской карьере, и «экземпляр» этого типа – запись:

```
DECLARE
   TYPE book_info_rt IS RECORD (
      author books.author%TYPE,
      category VARCHAR2(100),
      total_page_count POSITIVE);

   steven_as_author book_info_rt;
```

Обратите внимание, что я объявляю запись на основе типа TYPE, не используя атрибут %ROWTYPE. Элемент book_info_rt уже является типом.

Работа с записями

Вне зависимости от того, каким способом была определена запись (на основе таблицы, курсора или явного использования оператора TYPE для записи), работа со всеми записями ведется одинаково. Вы можете работать с данными, хранящимися в записи, на уровне записи (запись воспринимается как единое целое), а также обращаться напрямую к ее отдельным полям.

Операции на уровне записи

Когда вы работаете на уровне записи, то какие бы то ни было ссылки на отдельные поля отсутствуют. На текущий момент PL/SQL поддерживает следующие операции над записями:

- Вы можете копировать содержимое одной записи в другую (в случае совместимости их структур, то есть у них должно быть одинаковое количество полей и одинаковые или взаимно преобразуемые типы данных).
- Вы можете присваивать записи значение NULL простым присваиванием.
- Вы можете определять и передавать запись как аргумент в списке параметров.
- Вы можете возвращать (оператором RETURN) запись из функции.

Операции на уровне записей можно применять к любым записям с совместимыми структурами. Другими словами, записи должны иметь одинаковое количество полей и одинаковые или взаимно преобразуемые типы данных, но они не обязаны быть одного типа. Предположим, что мы создали такую таблицу:

```
CREATE TABLE cust_sales_roundup (
   customer_id NUMBER (5),
   customer_name VARCHAR2 (100),
   total_sales NUMBER (15,2)
   );
```

Три записи, определенные подобным образом, имеют совместимые структуры, и я могу перемешивать и сопоставлять данные в этих записях:

```
DECLARE
   cust_sales_roundup_rec cust_sales_roundup%ROWTYPE;

   CURSOR cust_sales_cur IS SELECT * FROM cust_sales_roundup;
   cust_sales_rec cust_sales_cur%ROWTYPE;

   TYPE customer_sales_rectype IS RECORD
      (customer_id NUMBER(5),
       customer_name customer.name%TYPE,
       total_sales NUMBER(15,2)
      );
   prefererred_cust_rec customer_sales_rectype;
BEGIN
   -- Присвоить одну запись другой.
   cust_sales_roundup_rec := cust_sales_rec;
   prefererred_cust_rec := cust_sales_rec;
END;
```

Операции на уровне поля

Если вам необходимо получить доступ к полю записи (чтобы прочитать или изменить значение), необходимо использовать точечную нотацию, в точности как при ссылке на столбец определенной таблицы базы данных. Синтаксис ссылки на поле будет таким:

```
[имя_схемы.][имя_пакета.]имя_записи.имя_поля
```

Имя пакета должно быть указано только в том случае, если запись определяется в спецификации пакета, отличного от того, в котором вы работаете в настоящий момент. Имя схемы следует указывать лишь в том случае, если пакет принадлежит не той схеме, в которой вы компилируете свой код.

После того как вы определили поле с помощью точечной нотации, вы можете ссылаться на значение этого поля и изменять его в соответствии с обычными правилами, принятыми в PL/SQL.

Коллекции

Коллекция – это структура данных, которая похожа на список или одномерный массив. Коллекция – ближайший родственник традиционного массива в PL/SQL. Коллекции можно использовать для управления массивами данных.

Типы коллекций

Oracle поддерживает коллекции трех видов. Они имеют много общего, но обладают и специфическими характеристиками.

Ассоциативные массивы

Это одномерные неограниченные разреженные коллекции однородных элементов, которые доступны только в PL/SQL. Они назывались *PL/SQL-таблицами* в версии PL/SQL 2 и *индекс-таблицами* в версиях Oracle8 Database и Oracle8*i* Database (название объясняется тем, что при объявлении такой коллекции необходимо явно указать, что она индексируется номером строки). В версии Oracle9*i* Database появилось название *ассоциативные массивы*. Изменение названия было вызвано тем, что начиная с этой версии конструкция `INDEX BY` может использоваться для «ассоциирования» (индексирования) содержимого значениями `VARCHAR2` или `PLS_INTEGER`.

Вложенные таблицы

Это также одномерные неограниченные коллекции однородных элементов. Изначально они являются плотными (*dense*), но в результате удалений могут стать разреженными. Вложенные таблицы могут быть определены как в PL/SQL, так и в базе данных (например, в качестве столбца таблицы). Вложенные таблицы являются *мультимножествами*, что означает, что не определен никакой внутренний порядок для элементов вложенных таблиц.

Массивы VARRAY

Как и другие типы коллекций, массивы переменной длины (тип `VARRAY`, *variable-sized arrays*) также являются одномерными коллекциями однородных элементов. Отличие в том, что они всегда ограничены и никогда не бывают разреженными. Определяя тип `VARRAY`, вы обязаны указать максимальное количество элементов, которое он может содержать. Как и вложенные таблицы, массивы типа `VARRAY` могут использоваться как в PL/SQL, так и в базе данных. Но в отличие от вложенных таблиц, порядок элементов в массиве `VARRAY` сохраняется при сохранении и извлечении такого массива.

Работа с коллекциями

В разделе рассматриваются достаточно простые примеры для каждого из типов коллекций с разъяснением их основных характеристик.

Использование ассоциативного массива

В следующем примере объявлен тип – ассоциативный массив, затем объявлена коллекция этого типа. Заполняем коллекцию четырьмя строками данных, затем в цикле проходим по ней, выводя содержащиеся в ней строковые значения. Более подробное описание будет приведено следом за фрагментом кода.

```
 1  DECLARE
 2     TYPE list_of_names_t IS TABLE OF person.first_name%TYPE
 3        INDEX BY PLS_INTEGER;
 4     happyfamily   list_of_names_t;
 5     l_row PLS_INTEGER;
 6  BEGIN
 7     happyfamily (2020202020) := 'Eli';
 8     happyfamily (-15070) := 'Steven';
 9     happyfamily (-90900) := 'Chris';
10     happyfamily (88) := 'Veva';
11
12     l_row := happyfamily.FIRST;
13
14     WHILE (l_row IS NOT NULL)
15     LOOP
16        DBMS_OUTPUT.put_line (happyfamily (l_row));
17        l_row := happyfamily.NEXT (l_row);
18     END LOOP;
19* END;
```

```
SQL> /
Chris
Steven
Veva
Eli
```

Строки	Описание
2–3	Объявляем тип – ассоциативный массив, используя специальное предложение `INDEX BY`. Коллекция, объявляемая на основе этого типа, содержит список строковых значений, длина каждого из которых определяется размерностью столбца `first_name` таблицы `person`.
4	Объявляем коллекцию `happyfamily` на основе типа `list_of_names_t`.
9–10	Заполняем коллекцию четырьмя именами. Имейте в виду, что можно было бы использовать практически любое целое значение. Номера строк ассоциативного массива не обязаны быть последовательными и могут быть даже отрицательными!
12	Вызываем метод (функцию, которая «привязана» к коллекции) `FIRST` для получения первого или наименьшего номера строки в коллекции.
14–18	Используем цикл `WHILE` для просмотра содержимого коллекции с выводом каждой строки. В строке 17 использован метод `NEXT` для перехода от текущей строки к следующей с пропуском возможных пустот.

Использование вложенной таблицы

В следующем примере сначала объявим тип – вложенную таблицу на уровне схемы данных. В PL/SQL-блоке объявим на основе этого типа три вложенных таблицы. Поместим во вложенную таблицу happyfamily имена всех членов семьи. Имена детей поместим во вложенную таблицу children. Затем используем появившийся в версии Oracle Database 10*g* оператор над множествами MULTISET EXCEPT для извлечения из вложенной таблицы happyfamily родителей и поместим их имена во вложенную таблицу parents. Выведем содержимое таблицы parents. Более подробное описание будет приведено следом за фрагментом кода.

```
REM Section A
SQL> CREATE TYPE list_of_names_t IS TABLE OF VARCHAR2 (100);
  2 /
Type created.

REM Section B
SQL>
  1 DECLARE
  2    happyfamily   list_of_names_t := list_of_names_t ();
  3    children      list_of_names_t := list_of_names_t ();
  4    parents       list_of_names_t := list_of_names_t ();
  5 BEGIN
  6    happyfamily.EXTEND (4);
  7    happyfamily (1) := 'Eli';
  8    happyfamily (2) := 'Steven';
  9    happyfamily (3) := 'Chris';
 10    happyfamily (4) := 'Veva';
 11
 12    children.EXTEND;
 13    children (1) := 'Chris';
 14    children.EXTEND;
 15    children (2) := 'Eli';
 16
 17    parents := happyfamily MULTISET EXCEPT children;
 18
 19    FOR l_row IN parents.FIRST .. parents.LAST
 20    LOOP
 21       DBMS_OUTPUT.put_line (parents (l_row));
 22    END LOOP;
 23* END;

SQL> /
Steven
Veva
```

Строки	Описание
Раздел A	Оператор CREATE TYPE создает тип – вложенную таблицу непосредственно в базе данных. Благодаря созданию типа в базе данных мы теперь имеем возможность объявлять вложенные таблицы в любом PL/SQL-

Строки	Описание
	блоке, который обладает полномочиями на SELECT для этого типа. Также можно объявлять на основе этого типа столбцы реляционных таблиц.
2–4	Объявляем три разных вложенных таблицы на основе определенного для схемы типа. Обратите внимание, что каждый раз для инициализации вложенной таблицы используется *функция-конструктор*. Эта функция всегда имеет такое же имя, что и тип, ее создает для нас Oracle. Для того чтобы можно было использовать вложенную таблицу, ее необходимо инициализировать.
6	Вызываем метод EXTEND для того, чтобы расширить коллекцию и вместить во вложенную таблицу членов моей семьи. Здесь, в отличие от ассоциативных массивов, необходимо явно запрашивать строку вложенной таблицы, прежде чем поместить в нее элемент.
7–10	Заполняем коллекцию happyfamily нашими именами.
12–15	Заполняем коллекцию children. В данном случае расширяем коллекцию построчно.
17	Для того чтобы определить, кто в этой семье является родителем, просто вычитаем children из happyfamily. В версиях, начиная с Oracle Database 10*g*, это очень легко сделать при помощи оператора над множествами MULTISET EXCEPT (он очень похож на SQL-команду MINUS).
19–22	Нам точно известно, что коллекция parents плотно заполнена в результате выполнения операции MULTISET EXCEPT, поэтому можно использовать цикл FOR со счетчиком для просмотра содержимого коллекции. Если попытаться использовать такую конструкцию для разреженной коллекции, то будет вызвано исключение NO_DATA_FOUND.

Использование VARRAY

Теперь рассмотрим использование массива VARRAY в качестве столбца реляционной таблицы. Сначала объявим два разных типа VARRAY на уровне схемы данных. Затем создадим реляционную таблицу family, в которой будет два столбца типа VARRAY. Наконец, в PL/SQL-программе заполним элементами две локальных коллекции и используем их в операторе INSERT для таблицы family. Более подробное описание будет приведено следом за фрагментом кода.

```
REM Section A
SQL> CREATE TYPE first_names_t IS VARRAY (2) OF VARCHAR2 (100);
  2  /
Type created.

SQL> CREATE TYPE child_names_t IS VARRAY (1) OF VARCHAR2 (100);
  2  /
Type created.

REM Section B
SQL> CREATE TABLE family (
```

```
  2    surname VARCHAR2(1000)
  3  , parent_names first_names_t
  4  , children_names child_names_t
  5  );

Table created.

REM Section C
SQL>
  1  DECLARE
  2     parents    first_names_t := first_names_t ();
  3     children   child_names_t := child_names_t ();
  4  BEGIN
  5     parents.EXTEND (2);
  6     parents (1) := 'Samuel';
  7     parents (2) := 'Charina';
  8     --
  9     children.EXTEND;
 10     children (1) := 'Feather';
 11
 12     --
 13     INSERT INTO family
 14                 (surname, parent_names, children_names
 15                 )
 16          VALUES ('Assurty', parents, children
 17                 );
 18  END;
SQL> /

PL/SQL procedure successfully completed.

SQL> SELECT * FROM family
  2  /

SURNAME
PARENT_NAMES
CHILDREN_NAMES
--------------------------------------------
Assurty
FIRST_NAMES_T('Samuel', 'Charina')
CHILD_NAMES_T('Feather')
```

Строки	Описание
Раздел A	Используем операторы CREATE TYPE для объявления двух разных типов VARRAY. Обратите внимание, что для типа VARRAY необходимо указывать максимальную длину коллекции. То есть, по сути, мое объявление в некоторой форме диктует определенную социальную политику: у человека должно быть не больше двух родителей и не больше одного ребенка.
Раздел B	Создаем реляционную таблицу с тремя столбцами: столбец VARCHAR2 для фамилии семьи и два столбца VARRAY: один для родителей, второй для детей.

Строки	Описание
Раздел C, строки 2–3	Объявляем две локальных переменных на основе типа VARRAY уровня схемы. Как и в случае с вложенными таблицами (и в отличие от ассоциативных массивов), необходимо использовать для инициализации структуры функцию-конструктор, имя которой совпадает с именем типа.
5–10	Расширяем коллекции и наполняем их именами родителей и единственного ребенка. Если попытаться выполнить расширение до двух строк, то Oracle инициирует исключение: «ORA-06532: Subscript outside of limit error».
13–17	Вставляем строку в таблицу family, просто указывая названия коллекций типа VARRAY в списке значений для таблицы. Несомненно, Oracle значительно облегчает для нас процедуру вставки коллекций в реляционные таблицы!

Встроенные методы коллекций

PL/SQL предлагает ряд встроенных процедур и функций, называемых *методами коллекций*, которые позволяют получать информацию о содержимом коллекций и изменять это содержимое. Доступны следующие методы коллекций:

Функция COUNT

Возвращает текущее количество элементов в коллекции.

Процедура DELETE

Удаляет из коллекции один или несколько элементов. Если элемент ранее не был удален, то уменьшает значение COUNT. Для коллекции типа VARRAY возможно удаление только всего содержимого коллекции сразу.

Функция EXISTS

Возвращает TRUE или FALSE в зависимости от того, существует ли определенный элемент.

Процедура EXTEND

Увеличивает количество элементов во вложенной таблице или коллекции типа VARRAY. Увеличивает значение COUNT.

Функции FIRST *и* LAST

Возвращают наименьший (FIRST) и наибольший (LAST) используемый индекс элемента коллекции.

Функция LIMIT

Возвращает максимально допустимое для коллекции типа VARRAY количество элементов.

Функции PRIOR *и* NEXT

Возвращают индекс элемента, непосредственно предшествующего (PRIOR) указанному или следующего за ним (NEXT). Эти методы все-

гда должны использоваться для обхода коллекций, особенно если речь идет о разреженных (или могущих такими стать) коллекциях.

Процедура `TRIM`

Удаляет элементы из конца коллекции (с наибольшим индексом). Уменьшает значение `COUNT`, если элементы не были ранее удалены посредством `DELETE`.

Эти программы называют *методами* потому, что при обращении к ним синтаксис отличается от обычного синтаксиса вызова процедур и функций. Методы коллекций используют *синтаксис методов*, который общепринят в объектно-ориентированных языках, таких как C++.

В общем виде вызов встроенных методов для ассоциативных массивов может иметь две формы:

- Операция без аргументов:

```
имя_таблицы.операция
```

- Операция, аргументом которой является индекс строки:

```
имя_таблицы.операция(номер_индекса [, номер_индекса])
```

Например, следующий оператор возвращает `TRUE` в случае, если определена 15-я строка ассоциативного массива `company_tab`:

```
company_tab.EXISTS(15)
```

Методы коллекций не доступны из SQL; они могут использоваться только в программах на PL/SQL.

Процедуры, функции и пакеты

PL/SQL поддерживает следующие структуры, позволяющие разбить ваш код на модули:

Процедура

Программа, которая выполняет одно или несколько действий и вызывается как исполняемый оператор PL/SQL. Используя список параметров, вы можете передавать информацию в процедуру и из нее.

Функция

Программа, которая возвращает единственное значение и используется как выражение PL/SQL. Используя список параметров, вы можете передавать информацию в функцию.

Пакет

Именованная коллекция процедур, функций, типов и переменных. На самом деле пакет является скорее не модулем, а метамодулем, но без его упоминания описание модульной структуры было бы неполным.

Триггер базы данных

Набор команд, который запускается на исполнение при наступлении определенного события в базе данных (например, вход в приложение, изменение строки таблицы, исполнение DDL-команды).

Объектный тип или экземпляр объектного типа

Oracle-версия объектно-ориентированного класса (то есть попытка его эмуляции). Объектные типы описывают как состояние, так и поведение, объединяя собственно данные (подобно реляционной таблице) с правилами (процедурами и функциями, манипулирующими с этими данными).

В этом разделе мы поговорим о процедурах, функциях и пакетах. Триггеры будут описаны далее. Объектные типы не будут рассматриваться в этой главе, так как мало кто из разработчиков (и почти никто из администраторов баз данных) использует объектно-ориентированные возможности Oracle.

Процедуры

Процедура – это модуль, выполняющий одно или несколько действий. Вызов процедуры в PL/SQL представляет собой независимый исполняемый оператор, поэтому PL/SQL-блок может состоять только из вызова процедуры. Процедуры являются ключевыми составляющими модульного кода, которые обеспечивают группировку и возможность повторного использования программной логики.

Структура процедуры

PL/SQL-процедура имеет следующий формат:

```
PROCEDURE [схема.]имя [( параметр [, параметр ...] ) ]
   [AUTHID DEFINER | CURRENT_USER]
IS
   [операторы объявления]
BEGIN
   исполняемые операторы
[ EXCEPTION
     операторы обработки исключений]
END [имя];
```

где каждый элемент имеет соответствующее назначение:

схема

Имя схемы, которой принадлежит процедура (необязательный параметр). По умолчанию процедура принадлежит схеме текущего пользователя. Для создания процедуры в другой схеме текущему пользователю потребуются соответствующие привилегии.

имя

Имя процедуры, которое указывается сразу после ключевого слова `PROCEDURE`.

параметры

Необязательный список параметров, которые могут быть определены для передачи информации как в процедуру, так и из нее обратно в вызывающую программу.

AUTHID *предложение*

Определяет, с какими правами будет исполняться процедура: с правами ее владельца (создателя) или же с правами вызывающего пользователя. Принято говорить о двух моделях исполнения: *с правами владельца* и *с правами вызывающего*.

операторы объявления

Объявления локальных идентификаторов для данной процедуры. Если вы ничего не объявляете, то операторы IS и BEGIN будут следовать непосредственно друг за другом.

исполняемые операторы

Операторы, которые процедура исполняет при вызове. После ключевого слова BEGIN до ключевых слов END или EXCEPTION должен быть указан хотя бы один исполняемый оператор.

операторы обработки исключений

Необязательные обработчики исключений для процедуры. Если вы не обрабатываете явно никакие исключения, то пропустите ключевое слово EXCEPTION и завершите раздел исполнения ключевым словом END.

Вызов процедуры

Процедура вызывается как исполняемый оператор PL/SQL. Другими словами, вызов процедуры должен завершаться точкой с запятой (;) и исполняться до или после других (если они есть) операторов SQL или PL/SQL в исполняемом разделе PL/SQL-блока.

Для запуска процедуры apply_discount используются следующие операторы:

```
BEGIN
   apply_discount( new_company_id, 0.15 );  -- скидка 15%
END;
```

Если процедура не принимает параметров, можно вызывать ее, не используя скобки:

```
display_store_summary;
```

В версиях Oracle8*i* Database и старше можно включать в вызов процедуры пустые скобки, например:

```
display_store_summary( );
```

Функции

Функция – это модуль, возвращающий значение. В отличие от вызова процедуры, являющегося независимым исполняемым оператором, вызов функции может существовать только как часть исполняемого оператора (то есть он может быть, например, элементом выражения или значением, присваиваемым по умолчанию при объявлении переменной).

Функция возвращает значение, которое, естественно, относится к какому-то типу данных. Функция может использоваться в PL/SQL-операторе вместо выражения, имеющего тот же тип данных, что и возвращаемое функцией значение.

Функции чрезвычайно важны при создании модульных конструкций. Например, любое бизнес-правило или формула в вашем приложении должны быть помещены в функции. Любой запрос, возвращающий единственную строку, также следует определять в функции, с тем чтобы обеспечить простой и надежный способ его повторного использования.

Некоторые разработчики предпочитают полагаться не на функции, а на процедуры, возвращающие информацию через список параметров. Если вы относитесь к их числу, то не забудьте проверить, что все ваши бизнес-правила, формулы и однострочные запросы «спрятаны» в процедуры.

Если в приложении определено и используется мало функций, то его, скорее всего, будет сложно поддерживать и совершенствовать.

Структура функции

Структура функции совпадает со структурой процедуры, единственное отличие состоит в том, что функция включает в себя еще предложение RETURN:

```
FUNCTION [схема.]имя [( параметр [, параметр ...] ) ]
   RETURN тип_возвращаемых_данных
   [AUTHID DEFINER | CURRENT_USER]
   [DETERMINISTIC]
   [PARALLEL ENABLE ...]
   [PIPELINED]
IS
   [операторы объявления]

BEGIN
   исполняемые операторы

[EXCEPTION
   операторы обработки исключений]

END [ имя ];
```

где элементы имеют следующее назначение:

схема

Имя схемы, которой принадлежит функция (необязательный параметр). По умолчанию функция принадлежит схеме текущего пользователя. Для создания процедуры в другой схеме текущему пользователю потребуются соответствующие привилегии.

имя

Имя функции, которое указывается сразу после ключевого слова FUNCTION.

параметры

Необязательный список параметров, которые могут быть определены для передачи информации как в функцию, так и из нее обратно в вызывающую программу.

тип_возвращаемых_данных

Тип данных значения, возвращаемого функцией, который обязательно должен быть указан в заголовке функции.

AUTHID *предложение*

Определяет, с какими правами будет исполняться процедура: с правами ее владельца (создателя) или же с правами вызывающего пользователя. Принято говорить о двух моделях исполнения: *с правами владельца* и *с правами вызывающего*.

DETERMINISTIC *предложение*

Подсказка оптимизатору, позволяющая системе использовать сохраненную копию возвращенного функцией результата (при его наличии). Оптимизатор запроса определяет, следует ли выбрать сохраненную копию или же вызвать функцию повторно.

PARALLEL_ENABLE *предложение*

Подсказка оптимизатору, разрешающая параллельное выполнение функции при вызове из оператора SELECT.

PIPELINED *предложение*

Указывает, что результаты табличной функции должны возвращаться построчно с помощью команды PIPE ROW.

операторы объявления

Объявления локальных идентификаторов для данной функции. Если вы ничего не объявляете, то операторы IS и BEGIN будут следовать непосредственно друг за другом.

исполняемые операторы

Операторы, которые функция исполняет при вызове. После ключевого слова BEGIN до ключевых слов END или EXCEPTION должен быть указан хотя бы один исполняемый оператор.

операторы обработки исключений

Необязательные обработчики исключений для функции. Если вы не обрабатываете явно никакие исключения, то пропустите ключевое слово EXCEPTION и завершите раздел исполнения ключевым словом END.

Вызов функции

Функция вызывается как часть исполняемого оператора и может использоваться в составе оператора в любом месте, где разрешено использование выражений. Рассмотрим на примерах различные варианты вызова функций:

- Присваивание переменной значения по умолчанию при помощи вызова функции:

```
DECLARE
   v_nickname VARCHAR2(100) :=
      favorite_nickname('Steven');
```

- Использование функции-члена для объектного типа pet в условном выражении:

```
DECLARE
   my_parrot pet_t :=
      pet_t (1001, 'Mercury', 'African Grey',
             TO_DATE ('09/23/1996', 'MM/DD/YYYY'));
BEGIN
   IF my_parrot.age < INTERVAL '50' YEAR -- тип INTERVAL в 9i
   THEN
      DBMS_OUTPUT.PUT_LINE ('Still a youngster!');
   END IF;
```

- Извлечение одной строки информации о книге непосредственно в запись:

```
DECLARE
   my_first_book books%ROWTYPE;
BEGIN
   my_first_book := book_info.onerow ('1-56592-335-9');
   ...
```

- Получение значения курсорной переменной с информацией о просроченных книгах для определенного пользователя:

```
DECLARE
   my_overdue_info overdue_rct;
BEGIN
   my_overdue_info :=
      book_info.overdue_info ('STEVEN_FEUERSTEIN');
   ...
```

Параметры

Процедуры и функции могут использовать *параметры* для передачи информации между модулем и вызывающим PL/SQL-блоком (в обоих направлениях).

Параметры модуля имеют не меньшее значение, чем реализующий модуль код (тело модуля). Конечно, вы должны обеспечить работоспособность модуля. Но все же суть создания модуля заключается в том, что он может вызываться (в идеале) несколькими другими модулями. Если же список параметров плохо составлен или непонятен, то другим программистам будет сложно использовать такой модуль, и мало кто захочет с ним возиться. И будет уже не важно, насколько хорошо вы написали свой модуль, если никто не захочет им пользоваться.

PL/SQL предлагает различные средства для эффективного составления списка параметров. В разделе будут рассмотрены все элементы определения параметров.

Определение параметров

Формальные параметры определяются в списке параметров программы. Определение параметра чрезвычайно похоже на объявление переменной в разделе объявлений блока PL/SQL. Два важных отличия состоят в следующем: во-первых, для параметра должен быть указан режим использования, а во-вторых, при определении параметров нельзя накладывать на них ограничения.

Объявление с ограничением – это объявление, накладывающее ограничение на значение, которое может присваиваться переменной, объявляемой с данным типом данных. *Объявление без ограничения* не накладывает таких ограничений на значения. В следующем объявлении на переменную `company_name` накладывается ограничение: ее длина не должна превышать 60 символов:

```
DECLARE
   company_name VARCHAR2(60);
```

Однако при определении параметра необходимо исключить часть объявления с ограничением:

```
PROCEDURE display_company (company_name IN VARCHAR2) IS ...
```

Фактические и формальные параметры

Необходимо различать две разновидности параметров: формальные и фактические. *Формальные параметры* – это имена, объявленные в списке параметров в заголовке модуля. *Фактические параметры* – это значения или выражения, помещенные в список параметров при фактическом вызове модуля.

Давайте посмотрим на отличия между формальными и фактическими параметрами на примере функции `tot_sales`. Вот ее заголовок:

```
FUNCTION tot_sales
   (company_id_in IN company.company_id%TYPE,
    status_in IN order.status_code%TYPE := NULL)
RETURN std_types.dollar_amount;
```

Формальными параметрами `tot_sales` являются:

`company_id_in`

Первичный ключ таблицы `company`.

`status_in`

Статус заказов, которые должны быть включены в расчет продаж.

Эти формальные параметры не существуют вне рамок функции. Вы можете воспринимать их как заполнители (placeholders) для фактических значений параметров, которые будут переданы в функцию, когда она будет использоваться в программе.

Каким образом при исполнении программы PL/SQL понимает, какой фактический параметр соответствует какому формальному? PL/SQL поддерживает два способа установки соответствия:

Связывание по позиции

Неявно связывает фактический параметр с формальным по их позиции.

Связывание по имени

Явно связывает фактический параметр с формальным по имени.

Связывание по позиции

Во всех рассмотренных ранее примерах использовалось связывание параметров по позиции. При связывании по позиции PL/SQL устанавливает соответствие между параметрами на основании их относительных позиций в списке: N-й фактический параметр в вызове программы сопоставляется N-му формальному параметру в заголовке программы.

В следующем примере для функции `tot_sales` PL/SQL связывает первый фактический параметр `:order.company_id` с первым формальным параметром `company_id_in`. Затем второй фактический параметр `N` сопоставляется второму формальному параметру `status_in`:

```
new_sales := tot_sales (:order.company_id, 'N');
```

```
FUNCTION tot_sales
   (company_id_in IN company.company_id%TYPE,
    status_in IN order.status_code%TYPE := NULL)
RETURN std_types.dollar_amount;
```

Теперь вы знаете, каким образом компилятор передает значения параметров в модули. Связывание по позиции, несомненно, является наиболее понятным и часто используемым методом.

Связывание по имени

При связывании по имени вы явно сопоставляете формальному параметру (имя параметра) фактический параметр (значение параметра) непосредственно в вызове программы, используя комбинацию символов =>.

Синтаксис связывания по имени будет таким:

```
имя_формального_параметра => значение_аргумента
```

Имя формального параметра явно указано, так что PL/SQL не должен пытаться установить соответствие фактического параметра формальному на основе их порядка. Так что если вы используете связывание по имени, то можете перечислять параметры в вызове программы не в том порядке, в котором указаны формальные параметры в заголовке. Например, вы можете вызвать функцию tot_sales для новых заказов любым из следующих двух способов:

```
new_sales :=
   tot_sales (company_id_in => order_pkg.company_id, status_in =>'N');

new_sales :=
   tot_sales (status_in =>'N', company_id_in => order_pkg.company_id);
```

Вы также можете использовать в одном вызове как связывание по имени, так и связывание по позиции:

```
:order.new_sales := tot_sales (order_pkg.company_id, status_in =>'N');
```

Если вы используете оба вида связывания, то все позиционные параметры должны предшествовать каким бы то ни было именованным (как это сделано в предыдущем примере).

Режимы использования параметров

Определяя параметры, вы должны указать, каким образом их можно использовать. Существуют три различных режима использования параметров.

Режим	Описание	Использование параметра
IN	Только для чтения	Значение параметра может использоваться внутри модуля, но изменение параметра запрещено.
OUT	Только для записи	Модуль может присвоить параметру значение, но ссылаться на него в модуле запрещено.
IN OUT	Для чтения и записи	Модуль может ссылаться на параметр (читать) и изменять его значение (записывать).

Режим использования определяет, каким образом программа может применять значение, присвоенное формальному параметру. Режим использования параметра задается непосредственно после указания имени параметра перед указанием его типа и значения по умолчанию

(если оно определено). В заголовке следующей процедуры указаны все три режима использования параметров:

```
PROCEDURE predict_activity
   (last_date_in IN DATE,
    task_desc_inout IN OUT VARCHAR2,
    next_date_out OUT DATE)
```

Процедура `predict_activity` принимает два элемента данных: дату последнего действия и описание этого действия. В результате работы процедура возвращает также два элемента данных: описание действия (возможно, измененное) и дату следующего действия. Параметр `task_desc_inout` определен как `IN OUT`, поэтому программа может читать аргумент и изменять его значение.

Пакеты

Пакет – это сгруппированные вместе элементы PL/SQL-кода. Пакеты представляют собой физическую и логическую структуру для организации программ и других элементов PL/SQL, таких как курсоры, типы и переменные. Они также предоставляют важные функциональные средства, такие как сокрытие логики и данных, определение глобальных данных (существующих на протяжении сеанса) и работу с ними.

Правила построения пакетов

Конструкция пакета очень проста. На изучение основных элементов синтаксиса пакета и основных правил его построения потребуется совсем немного времени, но многие недели (или даже месяцы) могут уйти на то, чтобы открыть для себя все нюансы структуры пакета. В этом разделе мы рассмотрим правила, которым необходимо следовать при создании пакета.

Для построения пакета необходимо создать его спецификацию и, почти всегда, тело пакета. Необходимо решить, какие элементы попадут в спецификацию, а какие будут скрыты в теле пакета. Можно также написать блок кода, который Oracle будет использовать для инициализации пакета.

Спецификация пакета

В спецификации пакета перечислены все доступные для использования в приложениях элементы пакета, а также приведена вся информация, которая необходима разработчику для использования этих элементов (эту информацию часто называют программным интерфейсом приложения (API – application programming interface)). Разработчик должен иметь возможность работать с элементами, приведенными в спецификации, не обращаясь за разъяснениями по их использованию к коду тела пакета.

Приведем некоторые правила создания спецификации пакета:

- Вы можете объявлять элементы практически всех типов данных: числа, исключения, типы и коллекции, на уровне пакета (то есть не внутри конкретной процедуры или функции пакета). Такие данные называются *данными уровня пакета*. При этом следует избегать объявления переменных в пакете, тогда как объявление констант вполне допустимо.

  В спецификации (или теле) пакета нельзя объявлять курсорные переменные (переменные, определяемые на основе типа `REF CURSOR`). Курсорные переменные не могут *сохранять свое значение* на протяжении сеанса (в разделе «Данные пакета» далее в этой главе будет приведена более подробная информация о длительности существования данных).
- Вы можете объявлять практически любые виды структур данных, такие как коллекции, записи или тип `REF CURSOR`.
- Вы можете объявлять в спецификации пакета процедуры и функции, но указывать можно только заголовок программы (все, что находится выше ключевого слова `IS` или `AS`).
- Вы можете включать в спецификацию пакета явные курсоры. Возможны две формы явного курсора: объявление запроса может включать в себя SQL-запрос, или же запрос может быть спрятан в теле пакета, тогда в объявлении курсора будет присутствовать только предложение `RETURN`.
- Если в спецификации пакета объявлены какие-то процедуры или функции, а также если курсорная переменная объявлена без SQL-запроса, то *необходимо* написать тело пакета, в котором будут реализованы эти элементы кода.
- В спецификацию пакета можно включить предложение `AUTHID`, которое будет определять, в соответствии с какими привилегиями разрешены любые ссылки на объекты: владельца пакета (`AUTHID DEFINER`) или пользователя, вызывающего пакет (`AUTHID CURRENT_USER`).
- После оператора `END` в спецификации пакета можно поместить необязательную метку с именем пакета, например:

```
END my_package;
```

Рассмотрим эти правила на примере очень простой спецификации пакета:

```
1  CREATE OR REPLACE PACKAGE favorites_pkg
2     AUTHID CURRENT_USER
3  IS
4     -- Две константы. Обратите внимание на то,
5     -- что им даны «говорящие» имена.
6
7     c_chocolate CONSTANT PLS_INTEGER := 16;
8     c_strawberry CONSTANT PLS_INTEGER := 29;
9
```

```
10      -- Объявление типа вложенной таблицы
11      TYPE codes_nt IS TABLE OF INTEGER;
12
13      -- Вложенная таблица объявляется на основе этого типа.
14      my_favorites codes_nt;
15
16      -- Тип REF CURSOR, возвращающий информацию из таблицы favorites.
17      TYPE fav_info_rct IS REF CURSOR RETURN favorites%ROWTYPE;
18
19      -- Процедура, которая получает список популярных товаров
20      -- (используя определенный ранее тип) и выводит
21      -- соответствующую информацию.
22      PROCEDURE show_favorites (list_in IN codes_nt);
23
24      -- Функция, которая возвращает всю информацию
25      -- из таблицы favorites о наиболее популярном элементе.
26      FUNCTION most_popular RETURN fav_info_rct;
27
28  END favorites_pkg; -- Метка конца для пакета
```

Как видите, спецификация пакета по структуре очень похожа на секцию объявлений PL/SQL-блока. Однако важным отличием является то, что спецификация пакета *не* может содержать никакого кода реализации.

Тело пакета

Тело пакета содержит весь код, который необходим для реализации спецификации пакета. Тело пакета требуется не всегда, но оно обязано присутствовать при наличии хотя бы одного из перечисленных далее условий:

Спецификация пакета содержит объявление курсора с предложением `RETURN`

Необходимо определить в теле пакета оператор `SELECT`.

Спецификация пакета содержит объявление процедуры или функции

Необходимо представить полную реализацию модуля в теле пакета.

Вам требуется исполнение кода в разделе инициализации тела пакета

Спецификация пакета не включает в себя исполняемый раздел (исполняемые операторы внутри конструкции `BEGIN...END`); вы можете исполнять код только в теле пакета.

По своей структуре тело пакета очень похоже на определение процедуры, но имеет и некоторые отличительные особенности:

- Тело пакета может включать в себя раздел объявлений, исполняемый раздел и раздел исключений. Раздел объявлений содержит полную реализацию всех курсоров и программ, определенных в спецификации, а также определение любых приватных элементов (не приведенных в спецификации). При наличии раздела инициализации раздел объявлений может быть пуст.

- Исполняемый раздел пакета принято называть *разделом инициализации*. Он может содержать код, который исполняется при создании в сеансе экземпляра пакета.[1]
- Раздел исключений обрабатывает любые исключения, порожденные в разделе инициализации. Он может присутствовать в конце пакета только в случае наличия раздела инициализации.
- Тело пакета может быть составлено следующими способами: оно может включать в себя только раздел объявлений, только исполняемый раздел, исполняемый раздел и раздел исключений, а также раздел объявлений, исполняемый раздел и раздел исключений.
- Не разрешается использовать предложение `AUTHID` в теле пакета, оно должно содержаться в спецификации. В теле пакета могут использоваться любые объекты, объявленные в спецификации этого пакета.
- Все правила и ограничения, существующие для объявления структур данных уровня пакета, относятся как к спецификации, так и к телу пакета. Например, запрещено объявлять курсорные переменные.
- После оператора `END` в спецификации пакета можно поместить необязательную метку с именем пакета, например:

```
END my_package;
```

Рассмотрим реализацию тела пакета `favorites_pkg`:

```
CREATE OR REPLACE PACKAGE BODY favorites_pkg
IS
   -- Приватная переменная
   g_most_popular   PLS_INTEGER := c_strawberry;

   -- Реализация процедуры
   PROCEDURE show_favorites (list_in IN codes_nt) IS
   BEGIN
      FOR indx IN list_in.FIRST .. list_in.LAST
      LOOP
         DBMS_OUTPUT.put_line (list_in (indx));
      END LOOP;
   END show_favorites;

   -- Реализация функции
   FUNCTION most_popular RETURN fav_info_rct
   IS
      retval fav_info_rct;
      null_cv fav_info_rct;
   BEGIN
      OPEN retval FOR
      SELECT *
        FROM favorites
```

[1] При первом обращении к пакету. – *Примеч. науч. ред.*

```
        WHERE code = g_most_popular;
      RETURN retval;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN RETURN null_cv;
   END most_popular;

END favorites_pkg; -- Метка окончания пакета
```

Правила вызова элементов пакета

На самом деле совершенно неважно знать, как запускается и исполняется пакет (в конце концов, это просто контейнер для элементов кода). Для пользователя важно знать, как запустить и использовать конкретные элементы, которые помещены в пакет.

Пакет является владельцем своих объектов точно так же, как таблица – владельцем своих столбцов. Чтобы обратиться к элементу, определенному в спецификации пакета, *извне* этого пакета, необходимо указать полное имя элемента, используя точечную нотацию. Давайте рассмотрим несколько примеров.

В данной спецификации пакета объявляются константа, исключение, курсор и несколько модулей:

```
CREATE OR REPLACE PACKAGE pets_inc
IS
   max_pets_in_facility CONSTANT INTEGER := 120;
   pet_is_sick EXCEPTION;

   CURSOR pet_cur (pet_id_in IN pet.id%TYPE) RETURN pet%ROWTYPE;

   FUNCTION next_pet_shots (pet_id_in IN pet.id%TYPE) RETURN DATE;
   PROCEDURE set_schedule (pet_id_in IN pet.id%TYPE);

END pets_inc;
```

Для ссылки на любой из этих объектов необходимо использовать имя пакета в качестве префикса перед именем объекта, а именно:

```
DECLARE
   -- Объявляем константу на основе столбца id таблицы pet.
   c_pet CONSTANT pet.id%TYPE:= 1099;
   v_next_apppointment DATE;
BEGIN
   IF pets_inc.max_pets_in_facility > 100
   THEN
      OPEN pets_inc.pet_cur (c_pet);
   ELSE
      v_next_appointment:= pets_inc.next_pet_shots (c_pet);
   END IF;
EXCEPTION
   WHEN pets_inc.pet_is_sick
   THEN
      pets_inc.set_schedule (c_pet);
END;
```

Подытоживая, можно сказать, что существуют два правила использования элементов пакета:

- При ссылке извне (во внешней программе) на элемент, определенный в спецификации пакета, необходимо использовать точечную нотацию в формате *имя_пакета.имя_элемента*.
- При ссылке внутри пакета (в спецификации или в теле) на элемент пакета указывать имя пакета не обязательно; PL/SQL автоматически интерпретирует ссылку как направленную на внутренний элемент пакета.

Данные пакета

Данные пакета представляют собой переменные и константы, которые определены *на уровне пакета* (то есть не внутри какой-то функции или процедуры пакета). Областью действия данных пакета является не какая-то одна программа, а весь пакет. Структуры данных пакета существуют (*сохраняют свои значения*) на протяжении всего сеанса, а не только во время исполнения одной программы.

Если данные пакета определены в теле пакета, то они сохраняют свои значения на протяжении сеанса, но доступ к ним разрешен только для элементов этого пакета (приватные данные).

Если данные пакета определены в спецификации пакета, то они сохраняют свои значения на протяжении сеанса и напрямую доступны (как на чтение, так и на запись) любой программе, обладающей привилегией EXECUTE для данного пакета (общие данные).

Если процедура пакета открывает курсор, то он остается открытым и доступным на протяжении всего сеанса. Нет необходимости в определении курсора в каждой программе. Один модуль может открывать курсор, а другой – выполнять выборку. Кроме того, переменные пакета могут передавать данные из одной транзакции в другую, так как переменные привязаны к сеансу, а не к какой-то определенной транзакции.

Выборка данных

Программы на PL/SQL запрашивают информацию из базы данных при помощи SQL-оператора SELECT. PL/SQL так тесно интегрирован с SQL, что вы можете исполнять SELECT прямо в PL/SQL-блоке, например:

```
DECLARE
   l_employee employee%ROWTYPE;
BEGIN
   SELECT * INTO l_employee
     FROM employee
    WHERE employee_id = 7500;
END;
```

Использованный в примере оператор SELECT INTO является примером *неявного курсора* и демонстрирует один из способов запроса данных из

PL/SQL-блока. Пользователю доступны следующие способы выполнения запроса к базе данных:

Неявные курсоры

Простой оператор `SELECT...INTO` извлекает одну строку данных непосредственно в переменные локальной программы. Это удобный (и часто наиболее эффективный) способ доступа к данным, использование которого, однако, может приводить к необходимости повторного кодирования оператора `SELECT` (или похожих операторов) в нескольких местах программы.

Явные курсоры

Вы можете явно объявить курсор в разделе объявлений (локального блока или пакета). В этом случае курсор можно будет открывать и извлекать данные в одной или нескольких программах, причем возможности контроля будут шире, чем при использовании неявных курсоров.

Курсорные переменные

Дополнительный уровень гибкости обеспечивают курсорные переменные (объявленные на основе типа `REF CURSOR`), которые позволяют передавать *указатель* на результирующее множество, полученное по запросу, из одной программы в другую. Любая программа, имеющая доступ к такой переменной, сможет открывать и закрывать курсор, а также выбирать из него данные.

Курсорные выражения

Появившиеся в версии Oracle9*i* Database выражения `CURSOR` преобразуют оператор `SELECT` в указатель (типа `REF CURSOR`) на результирующее множество и могут использоваться в сочетании с табличными функциями (о которых мы поговорим в главе 3) для повышения производительности приложений.

Подробно курсоры будут рассмотрены в главе 2.

Типичные операции над запросами

Для исполнения оператора SQL внутри программы PL/SQL выполняет одни и те же операции для всех типов курсоров. В одних случаях PL/SQL выполняет их автоматически, а в других (для явных курсоров) программисту необходимо написать соответствующий код.

Синтаксический анализ

Первым этапом обработки оператора SQL является его синтаксический анализ, который проводится для проверки корректности оператора и определения плана его выполнения.

Связывание

Связывание – это сопоставление значений из вашей программы (хост-переменных) заполнителям используемого оператора SQL.

Для статического SQL такое связывание выполняет само ядро PL/SQL. Для динамического SQL программист, если он планирует использовать переменные связывания, должен явно запросить выполнение этой операции.

Открытие

При открытии курсора переменные связывания используются для определения результирующего множества команды SQL. Указатель активной (текущей) строки устанавливается на первой строке. В некоторых случаях явное открытие курсора не требуется; ядро PL/SQL само выполняет эту операцию (например, для неявных курсоров или встроенного динамического SQL).

Исполнение

На этапе исполнения оператор выполняется внутри ядра SQL.

Выборка

При выполнении запроса команда `FETCH` извлекает следующую строку из результирующего множества курсора. При каждой выборке PL/SQL передвигает курсор вперед на одну строку по результирующему множеству. При работе с явными курсорами следует помнить, что в случае, когда строк для извлечения больше нет, `FETCH` ничего не делает (не инициирует исключение).

Закрытие

На этом этапе курсор закрывается, освобождается используемая им память. После закрытия курсор уже не содержит результирующее множество. В некоторых случаях явное закрытие курсора не требуется, ядро PL/SQL само выполняет эту операцию (например, для неявных курсоров или встроенного динамического SQL).

Атрибуты курсоров

PL/SQL поддерживает ряд *атрибутов курсоров*, которые можно применять для получения информации о состоянии курсора (табл. 1.5). При попытке ссылки на один из этих атрибутов для курсора, который еще не был открыт, Oracle обычно инициирует исключение `INVALID_CURSOR`.

Таблица 1.5. Атрибуты курсоров

Имя	Описание
`%FOUND`	Возвращает `TRUE`, если данные были выбраны, в противном случае – `FALSE`.
`%NOTFOUND`	Возвращает `TRUE`, если не выбрано ни одной строки, в противном случае – `FALSE`.
`%ROWCOUNT`	Возвращает количество строк, выбранных из курсора на текущий момент времени.

Имя	Описание
%ISOPEN	Возвращает TRUE, если курсор открыт, в противном случае – FALSE.
%BULK_ROWCOUNT	Возвращает количество строк, измененных командой FORALL, для каждого элемента коллекции.
%BULK_EXCEPTIONS	Возвращает информацию об исключении для строк, измененных командой FORALL, для каждого элемента коллекции.

Чтобы сослаться на атрибут курсора, укажите его после имени курсора или курсорной переменной, информацию о которых необходимо получить. Рассмотрим несколько примеров.

- Открыт ли все еще явный курсор?

```
DECLARE
   CURSOR happiness_cur IS SELECT simple_delights FROM ...;
BEGIN
   OPEN happiness_cur;
   ...
   IF happiness_cur%ISOPEN THEN ...
```

- Сколько строк было извлечено неявным пакетным запросом? (Обратите внимание, что «именем» курсора в данном случае является «SQL».)

```
DECLARE
   TYPE id_nt IS TABLE OF department.department_id;
   deptnums   id_nt;
BEGIN
   SELECT department_id
     BULK COLLECT INTO deptnums
     FROM department;

   DBMS_OUTPUT.PUT_LINE (SQL%BULK_ROWCOUNT);
END;
```

Вы можете ссылаться на атрибуты курсора в PL/SQL-коде (как это показано в примерах), но не можете использовать эти атрибуты внутри оператора SQL. Например, если бы вы попытались использовать атрибут %ROWCOUNT в предложении WHERE оператора SELECT, то результат был бы таким:

```
SELECT caller_id, company_id
FROM caller WHERE company_id = company_cur%ROWCOUNT;
```

компилятор выдаст сообщение об ошибке:

```
PLS-00229: Attribute expression within SQL expression
```

Неявные курсоры

PL/SQL объявляет неявный курсор и работает с ним каждый раз, когда вы выполняете DML-оператор SQL (INSERT, UPDATE или DELETE) или оператор SELECT INTO, который возвращает одну строку из базы данных и за-

писывает ее непосредственно в структуру данных PL/SQL. Такие курсоры называются *неявными*, потому что Oracle неявно (автоматически) выполняет множество операций обслуживания курсоров, таких как выделение памяти курсору, открытие курсора, выборка данных и т. д.

Неявный курсор – это оператор SELECT, обладающий следующими характеристиками:

- Оператор SELECT используется в исполняемом разделе блока и, в отличие от явных курсоров, не объявляется в разделе объявлений.
- Запрос включает в себя предложение INTO (или BULK COLLECT INTO для пакетной обработки). Предложение INTO является частью языка PL/SQL (а не SQL) и служит механизмом передачи данных из базы данных в локальные структуры данных PL/SQL.
- Не требуется открывать оператор SELECT, выбирать из него данные и закрывать его, все это происходит автоматически.

Неявный курсор имеет такой формат:

```
SELECT список столбцов
  [BULK COLLECT] INTO список переменных PL/SQL
  ...оставшаяся часть оператора SELECT...
```

Oracle выполняет открытие неявного курсора, выборку из него данных и закрытие автоматически; эти действия не подконтрольны программисту. Однако вы можете получить информацию о последней выполненной команде SQL, проанализировав значения атрибутов неявного курсора (об этом мы поговорим ниже в разделе «Атрибуты неявного курсора SQL»).

В последующих разделах под *неявным курсором* мы будем понимать оператор SELECT INTO, извлекающий (или пытающийся извлечь) одну строку данных. Позже будет рассмотрен оператор SELECT BULK COLLECT INTO, который позволяет извлекать несколько строк данных посредством одного неявного курсора.

Приведем пример неявного курсора, который извлекает строку и помещает ее в запись:

```
DECLARE
   l_book book%ROWTYPE;
BEGIN
   SELECT *
     INTO l_book
     FROM book
    WHERE isbn = '0-596-00121-5';
```

Обработка ошибок для неявных курсоров

Неявный курсор, реализуемый оператором SELECT, – это некое подобие «черного ящика». Вы передаете команду SQL в базу данных и получаете назад одну строку данных. Об отдельных операциях, таких как от-

крытие, извлечение данных и закрытие, вы ничего не знаете. Вам также придется иметь в виду, что Oracle автоматически инициирует исключения для неявного курсора SELECT в следующих двух случаях:

- Запрос не находит ни одной строки, соответствующей его условиям. В этом случае Oracle инициирует исключение NO_DATA_FOUND.
- Команда SELECT возвращает несколько строк. В этом случае Oracle инициирует исключение TOO_MANY_ROWS.

Атрибуты неявного курсора SQL

Oracle обеспечивает возможность получения информации о последнем выполненном неявном курсоре посредством ссылки на специальные атрибуты неявных курсоров (табл. 1.6, в которой также описаны значения, возвращаемые данными атрибутами для неявного SQL-запроса SELECT INTO). Курсоры являются неявными, имен у них нет, поэтому для их обозначения используется ключевое слово «SQL». Неявные курсоры также создаются для DML-операторов INSERT, UPDATE и DELETE.

Таблица 1.6. Атрибуты неявных курсоров и их значения

Имя	Описание
SQL%FOUND	Возвращает TRUE, если была выбрана или изменена одна строка (или несколько строк при работе с BULK COLLECT INTO), и FALSE – в противном случае (в этом случае Oracle также инициирует исключение NO_DATA_FOUND).
SQL%NOTFOUND	Возвращает TRUE, если оператором DML не было выбрано или изменено ни одной строки, и FALSE – в противном случае.
SQL%ROWCOUNT	Возвращает количество строк, выбранных или измененных курсором. Для курсора SELECT INTO будет иметь значение 1, если строка найдена, и 0, если Oracle инициирует исключение NO_DATA_FOUND.
SQL%ISOPEN	Всегда возвращает FALSE для неявных курсоров, так как Oracle открывает и закрывает неявные курсоры автоматически.

Все атрибуты неявных курсоров возвращают NULL, если неявные курсоры в текущем сеансе не выполнялись. В противном случае значения атрибутов всегда относятся к оператору SQL, выполненному последним, вне зависимости от того, в каком блоке или программе такой оператор выполнялся.

Явные курсоры

Явный курсор – это оператор SELECT, который явно определен в секции объявлений кода; такому курсору присваивается имя. Явные курсоры для операторов INSERT, UPDATE и DELETE не создаются.

При работе с явными курсорами программист обладает полным контролем над различными действиями PL/SQL, выполняемыми в ходе

извлечения информации из базы данных. Программист решает, когда следует открыть курсор (OPEN), когда выбирать из него записи (из таблицы или таблиц, указанных в команде SELECT данного курсора), сколько записей извлекать и когда закрывать курсор (CLOSE). Информацию о текущем состоянии курсора можно получить через его атрибуты. Возможность столь подробного поэтапного контроля делает явный курсор незаменимым средством программирования.

Давайте рассмотрим в качестве примера функцию, которая определяет (и возвращает) ту степень зависти, которую я испытываю к своим друзьям в зависимости от их места жительства:

```
 1  CREATE OR REPLACE FUNCTION jealousy_level (
 2     NAME_IN   IN   friends.NAME%TYPE) RETURN NUMBER
 3  AS
 4     CURSOR jealousy_cur
 5     IS
 6        SELECT location FROM friends
 7         WHERE NAME = UPPER (NAME_IN);
 8
 9     jealousy_rec   jealousy_cur%ROWTYPE;
10     retval         NUMBER;
11  BEGIN
12     OPEN jealousy_cur;
13
14     FETCH jealousy_cur INTO jealousy_rec;
15
16     IF jealousy_cur%FOUND
17     THEN
18        IF jealousy_rec.location = 'PUERTO RICO'
19           THEN retval := 10;
20        ELSIF jealousy_rec.location = 'CHICAGO'
21           THEN retval := 1;
22        END IF;
23     END IF;
24
25     CLOSE jealousy_cur;
26
27     RETURN retval;
28  END;
```

Этот блок PL/SQL выполняет следующие действия над курсором:

Строки	Описание
4–7	Объявление курсора.
9	Объявление записи на основе этого курсора.
12	Открытие курсора.
14	Выборка из курсора одной строки данных.
16	Проверка атрибута курсора для определения того, найдена ли строка.

Строки	Описание
18–22	Анализ содержимого выбранной строки для определения уровня зависти.
25	Закрытие курсора.

Для использования явного курсора необходимо сначала объявить его в разделе объявлений вашего PL/SQL-блока или пакета:

```
CURSOR имя_курсора [ ( [ параметр [, параметр ...] ) ]
   [ RETURN спецификация_возврата ]
   IS SELECT_оператор
       [FOR UPDATE [OF [список_столбцов]];
```

где *имя_курсора* – имя курсора, *спецификация_возврата* – необязательное предложение RETURN для курсора, а *SELECT_оператор* – любой корректный SQL-оператор SELECT. Вы также можете передавать в курсор параметры, используя необязательный список параметров. Как только курсор объявлен, вы можете открывать его и выбирать из него данные.

Oracle поддерживает для явных курсоров тот же набор атрибутов, что и для неявных курсоров. Значения, которые атрибуты явных курсоров могут приобретать до и после выполнения указанных операций над курсорами, приведены в табл. 1.7.

Таблица 1.7. Значения атрибутов явных курсоров «до и после» выполнения операций над курсорами

	%FOUND	%NOTFOUND	%ISOPEN	%ROWCOUNT
Перед OPEN	Инициируется ORA-01001	Инициируется ORA-01001	FALSE	Инициируется ORA-01001
После OPEN	NULL	NULL	TRUE	0
Перед первой FETCH	NULL	NULL	TRUE	0
После первой FETCH	TRUE	FALSE	TRUE	1
Перед последующими FETCH	TRUE	FALSE	TRUE	1
После последующих FETCH	TRUE	FALSE	TRUE	Зависит от данных
Перед последней FETCH	TRUE	FALSE	TRUE	Зависит от данных
После последней FETCH	FALSE	TRUE	TRUE	Зависит от данных
Перед CLOSE	FALSE	TRUE	TRUE	Зависит от данных
После CLOSE	Исключение	Исключение	FALSE	Исключение

BULK COLLECT

В Oracle8*i* Database появилось новое мощное средство, повышающее эффективность запросов в PL/SQL: предложение BULK COLLECT. При помощи BULK COLLECT вы можете извлекать в явном или неявном запросе несколько строк данных за одно обращение к базе данных. Использование BULK COLLECT уменьшает количество переключений контекста между PL/SQL и SQL, тем самым снижая издержки на извлечение данных. Предложение имеет следующий синтаксис:

```
... BULK COLLECT INTO имя_коллекции[, имя_коллекции] ...
```

где *имя_коллекции* – параметр, определяющий коллекцию. При работе с BULK COLLECT необходимо учитывать несколько правил и ограничений:

- В версиях, предшествующих Oracle9*i* Database, BULK COLLECT может использоваться только со статическим SQL. В Oracle9*i* Database и Oracle Database 10*g* BULK COLLECT может применяться как для статического, так и для динамического SQL.
- Ключевые слова BULK COLLECT могут быть использованы в любом из следующих предложений: SELECT INTO, FETCH INTO и RETURNING INTO.
- Коллекции, которые указываются в предложении BULK COLLECT, могут хранить только скалярные значения (строки, числа и даты). Другими словами, невозможно извлечение строки данных в запись, являющуюся элементом другой коллекции.
- Ядро SQL автоматически инициализирует и расширяет коллекции, которые задаются в предложении BULK COLLECT. Заполнение начинается с индекса 1, элементы вставляются последовательно (плотно), любые определенные ранее элементы перезаписываются.
- Использование пакетной выборки SELECT...BULK COLLECT в операторе FORALL недопустимо.
- В случае, если не возвращено ни одной строки, SELECT...BULK COLLECT *не* инициирует исключения NO_DATA_FOUND. Вам необходимо проверить содержимое коллекции на предмет наличия в ней данных.
- Операция BULK COLLECT перед исполнением запроса делает пустой коллекцию, заданную в предложении INTO. Если запрос не возвращает строк, то метод COUNT этой коллекции будет возвращать 0.

Ограничение количества строк, извлекаемых при помощи BULK COLLECT

Для ограничения количества строк, извлекаемых предложением BULK COLLECT из базы данных, Oracle поддерживает предложение LIMIT, которое имеет такой синтаксис:

```
FETCH курсор BULK COLLECT INTO ... [LIMIT строки];
```

где параметр *строки* может быть любым литералом, переменной или выражением, возвращающим целое значение (в противном случае Oracle инициирует исключение VALUE_ERROR).

Использование LIMIT для BULK COLLECT чрезвычайно полезно, так как позволяет регулировать объем памяти, используемый программой для обработки данных. Предположим, например, что необходимо выбрать и обработать 10 000 строк данных. Вы *можете* использовать BULK COLLECT для извлечения всех этих строк и заполнения большой коллекции. Однако такой подход потребует использования большого объема памяти в глобальной области процесса (PGA) данного сеанса. Если этот код исполняется несколькими разными пользователями Oracle, то производительность приложения может быть значительно снижена из-за свопирования PGA.

Рассмотрим фрагмент кода, в котором предложение LIMIT используется в операторе FETCH, исполняемом внутри простого цикла. Обратите внимание, что после выборки данных производится проверка атрибута %NOTFOUND (с тем чтобы определить, удалось ли в этот раз выбрать строки).

```
DECLARE
   CURSOR allrows_cur IS SELECT * FROM EMPLOYEE;

   TYPE employee_aat IS TABLE OF allrows_cur%ROWTYPE
      INDEX BY BINARY_INTEGER;

   l_employees employee_aat;
   l_row PLS_INTEGER;
BEGIN
   OPEN allrows_cur;
   LOOP
      FETCH allrows_cur BULK COLLECT INTO l_employees
         LIMIT 100;
      EXIT WHEN allrows_cur%NOTFOUND;[1]

      -- Просмотр коллекции и обработка данных.

      l_row := l_employees.FIRST;
      WHILE (l_row IS NOT NULL)
      LOOP
         upgrade_employee_status (l_employees(l_row).employee_id);
         l_row := l_employees.NEXT (l_row);
      END LOOP;
   END LOOP;

   CLOSE allrows_cur;
END;
```

Курсорные переменные и типы REF CURSOR

Курсорная переменная – это переменная, указывающая или ссылающаяся на курсор. В отличие от неявного курсора, который определяет

[1] Использование условия EXIT WHEN allrows_cur%NOTFOUND, скорее всего, приведет к тому, что последний «пакет» строк не будет обработан. Правильно было бы использовать EXIT WHEN l_employees.COUNT = 0. – *Примеч. науч. ред.*

имя рабочей области для результирующего множества, курсорная переменная является ссылкой на эту рабочую область. Явный и неявный курсоры являются статическими в том смысле, что они связаны с конкретными запросами. Курсорная же переменная может открываться для любого запроса и даже для нескольких разных запросов в рамках исполнения одной программы.

Объявление типов REF CURSOR

Для работы с курсорной переменной необходимо объявить ее, что выполняется в два этапа:

1. Сначала следует создать тип курсора оператором TYPE.
2. Затем на основе созданного типа объявить фактическую курсорную переменную.

Синтаксис создания типа курсора, на основе которого объявляется курсорная переменная, следующий:

```
TYPE имя_типа_курсора IS REF CURSOR [ RETURN возвращаемый_тип ];
```

где *имя_типа_курсора* – имя типа курсора, а *возвращаемый_тип* – спецификация возвращаемых курсорным типом данных. В качестве *возвращаемого_типа* может использоваться любая структура данных, допустимая в обычном предложении RETURN для курсора, которая определена при помощи атрибута %ROWTYPE или посредством ссылки на ранее определенный тип записи.

Обратите внимание, что предложение RETURN в объявлении типа REF CURSOR не является обязательным. Допустимы оба приведенных далее объявления:

```
TYPE company_curtype IS REF CURSOR RETURN company%ROWTYPE;
TYPE generic_curtype IS REF CURSOR;
```

Первая форма объявления REF CURSOR называется объявлением *строго типизированного* типа, так как оно связывает тип курсорной переменной с типом записи (или типом строки таблицы) в момент объявления. Любая объявленная таким способом курсорная переменная может использоваться только с теми командами SQL и структурами данных FETCH INTO, которые соответствуют заданному типу записи. Преимущество строго типизированного объявления REF CURSOR заключается в том, что компилятор может определить, правильно ли разработчик установил соответствие инструкций FETCH для курсорной переменной списку запроса объекта курсора.

Вторая форма объявления REF CURSOR, в которой отсутствует предложение RETURN, называется объявлением *слабо типизированного* типа. Такому типу курсорной переменной не сопоставляются никакие структуры данных. Курсорные переменные, объявленные на основе типов, созданных без использования предложения RETURN, могут применяться более гибко. Их можно использовать для любых запросов, с любыми

структурами возвращаемых данных, которые могут изменяться в ходе исполнения одной и той же программы.

Начиная с версии Oracle9*i* Database поддерживается предопределенный слабо типизированный тип REF CURSOR – SYS_REFCURSOR. Теперь нет необходимости определять собственный слабо типизированный тип, можно воспользоваться имеющимся:

```
DECLARE
   my_cursor SYS_REFCURSOR;
```

Объявление курсорных переменных

Синтаксис объявления курсорной переменной таков:

```
имя_курсора имя_типа_курсора;
```

где *имя_курсора* – имя курсорной переменной, а *имя_типа_курсора* – имя определенного ранее посредством оператора TYPE типа курсора.

Рассмотрим пример создания курсорной переменной:

```
DECLARE
   /* Создаем тип курсора для спортивных автомобилей. */
   TYPE sports_car_cur_type IS REF CURSOR RETURN car%ROWTYPE;

   /* Создаем курсорную переменную для спортивных автомобилей. */
   sports_car_cur sports_car_cur_type;
BEGIN
   ...
END;
```

Открытие курсорных переменных

Вы присваиваете значение (курсорный объект) курсору при открытии (OPEN) этого курсора. Синтаксис традиционного оператора OPEN разрешает для курсорных переменных использование в предложении FOR оператора SELECT:

```
OPEN имя_курсора FOR оператор_SELECT;
```

где *имя_курсора* – это имя курсорной переменной, а *оператор_SELECT* – это SQL-оператор SELECT.

Для курсорных переменных строго типизированного типа REF CURSOR структура оператора SELECT (количество и тип данных столбцов) должна совпадать или быть совместимой со структурой, указанной в инструкции RETURN объявления типа. Если же курсорная переменная определена на основе слабо типизированного типа REF CURSOR, то ее можно открывать (OPEN) для любого запроса с любой структурой данных.

Выборка данных из курсорных переменных

Как уже говорилось, синтаксис инструкции FETCH для курсорной переменной совпадает с синтаксисом для статических курсоров:

```
FETCH имя_курсорной_переменной INTO имя_записи;
FETCH имя_курсорной_переменной INTO имя_переменной, имя_переменной ...;
```

Если курсорная переменная объявлена на основе строго типизированного типа REF CURSOR, то компилятор PL/SQL проверяет совместимость структур данных, перечисленных после ключевого слова INTO со структурой запроса, связанного с курсорной переменной.

Если же курсорная переменная относится к слабо типизированному типу REF CURSOR, то компилятор PL/SQL не может выполнить подобную проверку. Данные из такой курсорной переменной могут извлекаться в любую структуру данных, так как в момент своего объявления тип курсора не был связан с типом строки таблицы. При компиляции невозможно определить, какой объект курсора (и соответственно команда SQL) будет присвоен данной переменной.

Следовательно, проверку совместимости приходится выполнять в процессе исполнения FETCH. Если при этом оказывается, что структуры запроса и предложения INTO не совпадают, то исполняющее ядро PL/SQL инициирует предопределенное исключение ROWTYPE_MISMATCH. Следует иметь в виду, что при необходимости (и при наличии такой возможности) PL/SQL будет использовать неявные преобразования типов.

Изменение данных

Подробное описание возможностей DML-операторов языка Oracle SQL выходит за рамки нашей книги. Вниманию читателей будет предложен лишь краткий обзор основ синтаксиса DML-операторов, затем будут рассмотрены специальные вопросы, связанные с использованием DML в PL/SQL, а именно:

- Примеры всех операторов DML
- Атрибуты курсоров для операторов DML
- Специальные средства PL/SQL для операторов DML, например предложение RETURNING

Дополнительную информацию вы сможете получить в документации по Oracle или SQL.

В языке SQL поддерживаются три оператора DML:

INSERT

Вставляет в таблицу одну или несколько новых строк.

UPDATE

Обновляет значения в одном или нескольких столбцах существующей строки таблицы.

DELETE

Удаляет из таблицы одну или несколько строк.

Оператор INSERT

Рассмотрим синтаксис двух основных типов операторов INSERT:

- Вставка одной строки с явно заданным списком значений.

```
INSERT INTO таблица [(столбец1, столбец2, ..., столбецn)]
   VALUES (значение1, значение2, ..., значениеn);
```

- Вставка в таблицу одной или нескольких строк, получаемых в результате выполнения оператора SELECT для одной или нескольких таблиц.

```
INSERT INTO таблица [(столбец1, столбец2, ..., столбецn)]
AS
   SELECT ...;
```

Давайте рассмотрим несколько примеров выполнения операторов INSERT внутри PL/SQL-блока. Сначала вставим новую строку в таблицу book. Обратите внимание, что если указаны значения для всех столбцов таблицы, то нет необходимости в перечислении имен столбцов:

```
BEGIN
   INSERT INTO book
        VALUES ('1-56592-335-9',
           'Oracle PL/SQL Programming',
           'Reference for PL/SQL developers,' ||
           'including examples and best practice ' ||
           'recommendations.',
           'Feuerstein,Steven, with Bill Pribyl',
           TO_DATE ('01-SEP-1997','DD-MON-YYYY'),
           987);
END;
```

Можно указать имена столбцов, а в качестве значений задать не литералы, а переменные:

```
DECLARE
   l_isbn book.isbn%TYPE := '1-56592-335-9';
   ... other declarations of local variables
BEGIN
   INSERT INTO books (
        isbn, title, summary, author,
        date_published, page_count)
      VALUES (
         l_isbn, l_title, l_summary, l_author,
         l_date_published, l_page_count);
```

Оператор UPDATE

При помощи оператора UPDATE можно обновить значения одного или нескольких столбцов в одной или нескольких строках таблицы. Базовая конструкция оператора такова:

```
UPDATE таблица
   SET столбец1 = значение1
       [, столбец2 = значение2, ... столбецN = значениеN]
[WHERE предложение_WHERE];
```

Предложение WHERE является необязательным: если оно не задано, то обновляются все строки таблицы. Рассмотрим несколько примеров:

- Привести все названия книг в таблице book к верхнему регистру.

```
UPDATE books SET title = UPPER (title);
```

- Запустить процедуру, которая удаляет составляющую времени из даты публикации книг определенных авторов (имя автора является аргументом процедуры) и приводит названия этих книг к верхнему регистру. Как видите, оператор UPDATE может использоваться как автономно, так и внутри PL/SQL-блока:

```
CREATE OR REPLACE PROCEDURE remove_time (
   author_in IN VARCHAR2)
IS
BEGIN
   UPDATE books
      SET title = UPPER (title),
          date_published =
             TRUNC (date_published)
    WHERE author LIKE author_in;
END;
```

Оператор DELETE

Оператор DELETE можно использовать для удаления одной, нескольких или всех строк таблицы:

```
DELETE FROM таблица
 [WHERE предложение_WHERE];
```

Предложение WHERE в операторе DELETE является необязательным: если оно не задано, то удаляются все строки таблицы. Рассмотрим несколько примеров:

- Удаление всех книг из таблицы books:

```
DELETE FROM books;
```

- Удаление из таблицы books всех книг, которые были опубликованы до определенной даты, и возврат количества удаленных строк:

```
CREATE OR REPLACE PROCEDURE remove_books (
   date_in             IN       DATE,
   removal_count_out   OUT      PLS_INTEGER)
IS
BEGIN
   DELETE FROM books WHERE date_published < date_in;
   removal_count_out := SQL%ROWCOUNT;
END;
```

Конечно, при работе с реальными объектами все эти команды DML становятся существенно более сложными. Например, вы можете обновить несколько столбцов данными, полученными в результате выполнения подзапроса. Начиная с версии Oracle9*i* Database можно заменить имя таблицы *табличной функцией*, возвращающей результирующее множество, с которым и будет работать оператор DML (см. также главу 3).

Oracle поддерживает ряд атрибутов курсора для неявных курсоров, стоящих за операторами DML, – они будут рассмотрены в следующем разделе.

Атрибуты курсоров для операций DML

Oracle позволяет получить доступ к информации о последней выполненной встроенной команде DML через атрибуты неявных курсоров SQL (они совпадают с атрибутами, перечисленными в табл. 1.6). Значения, возвращаемые данными атрибутами для операторов DML, представлены в табл. 1.8.

Таблица 1.8. Атрибуты неявных курсоров SQL для операторов DML

Имя	Описание
SQL%FOUND	Возвращает TRUE, если была модифицирована (создана, изменена или удалена) одна или несколько строк.
SQL%NOTFOUND	Возвращает TRUE, если командой DML не было модифицировано ни одной строки.
SQL%ROWCOUNT	Возвращает количество строк, модифицированных командой DML.

Рассмотрим примеры использования атрибутов неявных курсоров.

- Используем атрибут SQL%FOUND для определения того, обработал ли наш оператор DML сколько-нибудь строк. Пусть, например, некоторый автор время от времени меняет свое имя и хочет, чтобы именно это новое имя использовалось во всех его книгах. Создаем маленькую процедуру, которая будет обновлять имя автора, а затем сообщать о том, были ли изменены какие-то строки, через логическую переменную:

```
CREATE OR REPLACE PROCEDURE change_author_name (
   old_name_in         IN       books.author%TYPE,
   new_name_in         IN       books.author%TYPE,
   changes_made_out    OUT      BOOLEAN)
IS
BEGIN
   UPDATE books
      SET author = new_name_in
    WHERE author = old_name_in;

   changes_made_out := SQL%FOUND;
END;
```

- Используем атрибут SQL%ROWCOUNT в случае, когда необходимо определить точное количество строк, обработанных оператором DML. Переработаем приведенную выше процедуру изменения имени, с тем чтобы она возвращала чуть больше информации:

```
CREATE OR REPLACE PROCEDURE change_author_name (
   old_name_in         IN       books.author%TYPE,
   new_name_in         IN       books.author%TYPE,
   rename_count_out    OUT      PLS_INTEGER)
IS
BEGIN
   UPDATE books
      SET author = new_name_in
    WHERE author = old_name_in;

   rename_count_out := SQL%ROWCOUNT;
END;
```

DML и обработка исключений

При возникновении исключения в PL/SQL-блоке Oracle не производит отмену (откат) изменений, внесенных командами DML этого блока. Выбор поведения приложения в подобной ситуации осуществляет тот, кто управляет логикой транзакций приложения, то есть сам программист. При этом необходимо учитывать следующие соображения:

- Если ваш блок является автономной транзакцией (мы поговорим о них чуть позже в этой же главе), то при возникновении исключения необходимо выполнить откат или фиксацию изменений (обычно выполняется откат).
- Для ограничения области отката можно использовать *точки сохранения*. Другими словами, можно выполнить откат изменений вплоть до некоторой точки сохранения, сохранив тем самым часть изменений, выполненных в рамках сеанса (подробно о точках сохранения мы также поговорим далее в этой главе).
- Если исключение распространяется за пределы самого внешнего блока (то есть остается необработанным), то в большинстве сред выполнения PL/SQL, таких как SQL*Plus, автоматически производится безусловный (unqualified) откат, и все сделанные ранее изменения отменяются.

Пакетные операции DML и оператор FORALL

В версии Oracle8*i* Database возможности по использованию DML в PL/SQL были значительно усовершенствованы за счет появления оператора FORALL. Оператор FORALL указывает исполняющему ядру PL/SQL на необходимость пакетного связывания в команде SQL всех элементов одной или нескольких коллекций перед их отправкой ядру SQL. Какая от этого может быть польза? Все мы знаем о том, что PL/SQL тесно

связан с ядром SQL базы данных Oracle. PL/SQL – это лучший язык программирования для баз данных Oracle (даже несмотря на то, что теперь вы можете, по крайней мере, теоретически, использовать в тех же целях и язык Java).

Но такая тесная интеграция не обязательно означает отсутствие накладных расходов при запуске SQL из PL/SQL-программы. Когда исполняющее ядро PL/SQL обрабатывает блок кода, оно исполняет процедурные команды внутри собственного ядра, а команды SQL пересылает ядру SQL, где они исполняются, и результат (при необходимости) возвращается в ядро PL/SQL.

Передача управления от ядра PL/SQL ядру SQL и обратно называется *переключением контекста*. Каждое такое переключение означает дополнительные издержки. В определенных ситуациях таких переключений может быть очень много, что существенно снижет производительность. В версии Oracle8*i* Database появляются две возможности объединения нескольких переключений контекста в одно, таким образом повышая производительность приложения. Речь идет об операторе FORALL и предложении BULK COLLECT (которое было рассмотрено ранее).

При пакетном связывании команды и передаче ее ядру SQL команда исполняется один раз для каждого индекса из диапазона. Другими словами, выполняются те же команды SQL, но все они выполняются в рамках одного обращения к ядру SQL, то есть происходит всего одно переключение контекста (рис. 1.4).

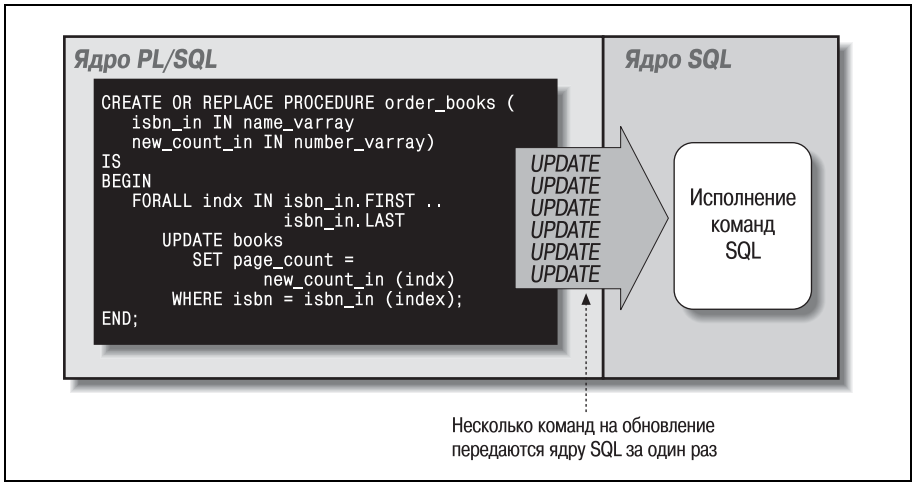


***Рис. 1.4.** Одно переключение контекста в операторе FORALL*

Синтаксис оператора FORALL

Несмотря на то что оператор FORALL включает в себя схему итерации (в соответствии с которой он перебирает элементы коллекции), он не

является циклом FOR. Следовательно, в нем нет ключевых слов LOOP и END LOOP. Синтаксис оператора FORALL таков:

```
FORALL индекс_строки IN
   [ нижняя_граница ... верхняя_граница |
     INDICES OF индексирующая_коллекция |
     VALUES OF индексирующая_коллекция
   ]
   [ SAVE EXCEPTIONS ]
   sql_команда;
```

где:

индекс_строки

Определяет коллекцию, элементы которой будет перебирать оператор FORALL.

нижняя_граница

Начальное значение индекса (строки или элемента коллекции) для выполнения операции.

верхняя_граница

Конечное значение индекса (строки или элемента коллекции) для выполнения операции.

sql_команда

Команда SQL, которая должна быть выполнена для каждого элемента коллекции.

индексирующая_коллекция

Коллекция PL/SQL, используемая для выбора индексов в связанном массиве, заданном в *sql_команде*. Возможность использования операторов INDICES_OF и VALUES_OF появилась в версии Oracle Database 10*g*.

SAVE EXCEPTIONS

Необязательное предложение, которое сообщает оператору FORALL о необходимости обработки всех строк с сохранением всех возникающих исключений.

При работе с FORALL необходимо соблюдать следующие правила:

- Телом оператора FORALL должна являться только одна команда DML: INSERT, UPDATE или DELETE.
- Оператор DML должен ссылаться на элементы коллекции посредством переменной *индекс_строки*, которая задана в операторе FORALL. Область действия переменной *индекс_строки* ограничивается оператором FORALL; вы не можете ссылаться на нее извне этого оператора. Однако обратите внимание, что верхняя и нижняя границы коллекций не обязательно должны охватывать все содержимое коллекций.

- Не следует объявлять переменную *индекс_строки*. Она объявляется неявно как переменная типа PLS_INTEGER ядром PL/SQL.
- Верхняя и нижняя границы должны задавать допустимый диапазон последовательных номеров индексов для коллекции (коллекций), заданных в команде SQL. Для разреженной коллекции будет сгенерирована следующая ошибка:

```
ORA-22160: element at index [3] does not exist
```

  Пример такой ситуации приведен в файле *diffcount.sql*, который можно найти на веб-сайте книги.

  Тем не менее Oracle Database 10*g* поддерживает операторы INDICES OF и VALUES OF для разреженных коллекций (в которых пропущены какие-то элементы).
- В операторе DML нельзя ссылаться на отдельные поля коллекций записей. Даже в случае, когда поле коллекции является коллекцией скаляров или коллекцией более сложных объектов, разрешено ссылаться только на строку коллекции целиком. Например, такой код:

```
DECLARE
   TYPE employee_aat IS TABLE OF employee%ROWTYPE
      INDEX BY PLS_INTEGER;
   l_employees   employee_aat;
BEGIN
   FORALL l_index IN l_employees.FIRST .. l_employees.LAST
      INSERT INTO employee (employee_id, last_name)
        VALUES (l_employees (l_index).employee_id
              , l_employees (l_index).last_name
        );
END;
```

  вызовет при компиляции такую ошибку:

```
PLS-00436: implementation restriction: cannot reference fields of BULK
In-BIND table of records
```

- Индекс элемента коллекции, заданной в операторе DML, не может быть выражением. Например, выполнение следующего фрагмента

```
DECLARE
   names name_varray := name_varray ();
BEGIN
   FORALL indx IN names.FIRST .. names.LAST
      DELETE FROM emp WHERE ename = names(indx+10);
END;
```

  вызовет появление такой ошибки:

```
PLS-00430: FORALL iteration variable INDX is not allowed in this context
```

Примеры FORALL

Рассмотрим несколько примеров использования оператора FORALL:

- Перепишем процедуру order_books с использованием FORALL:

```
CREATE OR REPLACE PROCEDURE order_books (
   isbn_in IN name_varray,
   new_count_in IN number_varray)
IS
BEGIN
   FORALL indx IN isbn_in.FIRST .. isbn_in.LAST
      UPDATE books
         SET page_count = new_count_in (indx)
       WHERE isbn = isbn_in (indx);
END;
```

Все изменения заключаются в замене FOR на FORALL и удалении ключевых слов LOOP и END LOOP. При этом FORALL передает команде SQL все строки, определенные в двух коллекциях. На рис. 1.4 показано, как организовано исполнение подобной процедуры.

- В следующем примере будет показано, как оператор DML может ссылаться на несколько коллекций. Возьмем три коллекции: denial, patient_name и illnesses. Проиндексированы элементы только двух первых коллекций, так что отдельные элементы этих коллекций будут переданы по индексу в каждую команду INSERT. Третьим столбцом таблицы health_coverage является коллекция, перечисляющая некоторые условия. Так как ядро PL/SQL выполняет пакетное связывание только для проиндексированных коллекций, то коллекция illnesses будет целиком помещена в третий столбец каждой вставляемой строки:

```
FORALL indx IN denial.FIRST .. denial.LAST
   INSERT INTO health_coverage
      VALUES (denial(indx), patient_name(indx), illnesses);
```

- Используем предложение RETURNING в операторе FORALL для извлечения информации о каждой отдельной команде DELETE. Следует помнить, что предложение RETURNING оператора FORALL должно использовать вложенное предложение BULK COLLECT INTO («пакетная» операция для запросов):

```
CREATE OR REPLACE FUNCTION remove_emps_by_dept (deptlist dlist_t)
   RETURN enolist_t
IS
   enolist enolist_t;
BEGIN
   FORALL aDept IN deptlist.FIRST..deptlist.LAST
      DELETE FROM emp WHERE deptno IN deptlist(aDept)
         RETURNING empno BULK COLLECT INTO enolist;
   RETURN enolist;
END;
```

- Используем индексы, определенные в одной коллекции, для определения того, какие строки из массива связывания (коллекции, за-

данной внутри команды SQL) должны использоваться динамическим оператором INSERT.

```
FORALL indx IN INDICES OF l_top_employees
   EXECUTE IMMEDIATE
      'INSERT INTO ' || l_table || ' VALUES (:emp_pky, :new_salary)
      USING l_new_salaries(indx).employee_id,
            l_new_salaries(indx).salary;
```

Управление транзакциями в PL/SQL

Как и следовало ожидать, реляционная база данных Oracle поддерживает очень мощный и надежный механизм транзакций. Код вашего приложения определяет, из чего будет состоять *транзакция* – логическая единица работы, результат которой сохраняется при помощи оператора COMMIT или отменяется оператором ROLLBACK. Транзакция неявно начинается с первого оператора SQL, выполненного после последнего оператора COMMIT или ROLLBACK (или с начала сеанса), или продолжается после ROLLBACK TO SAVEPOINT.

PL/SQL содержит ряд операторов для управления транзакциями:

COMMIT

Сохраняет все изменения, сделанные после последней операции COMMIT или ROLLBACK, и освобождает все блокировки.

ROLLBACK

Отменяет все изменения, сделанные после последней операции COMMIT или ROLLBACK, и освобождает все блокировки.

ROLLBACK TO SAVEPOINT

Отменяет все изменения, сделанные после установки указанной точки сохранения, и освобождает блокировки, которые были установлены в данном фрагменте кода.

SAVEPOINT

Устанавливает точку сохранения, которая затем позволит выполнять частичный откат.

SET TRANSACTION

Позволяет начать сеанс[1] в режиме «только для чтения» или «для чтения и записи», задать уровень изоляции или сопоставить текущей транзакции определенный сегмент отката.

LOCK TABLE

Позволяет заблокировать всю таблицу базы данных в определенном режиме. Позволяет изменить обычно применяемую к таблице установку по умолчанию – блокировку на уровне строк.

---

[1] На самом деле не сеанс, а конкретную транзакцию. – *Примеч. науч. ред.*

В последующих разделах мы рассмотрим операторы COMMIT и ROLLBACK, а также *автономные транзакции* PL/SQL.

Оператор COMMIT

Выполнение оператора COMMIT делает постоянными изменения, внесенные вашим сеансом в базу данных в рамках текущей транзакции. После выполнения COMMIT (фиксации транзакции) сделанные вами изменения станут видимыми для других сеансов и пользователей Oracle. Синтаксис оператора COMMIT:

```
COMMIT [WORK] [COMMENT текст];
```

Ключевое слово WORK является необязательным и может использоваться для улучшения читаемости.

Ключевое слово COMMENT служит для ввода комментария, относящегося к текущей транзакции. Текст комментария должен представлять собой заключенный в кавычки литерал длиной не более 50 символов. Комментарий обычно используется для распределенных транзакций и может оказаться полезным при исследовании и разрешении сомнительных транзакций при двухфазной фиксации. Комментарий хранится в словаре данных вместе с идентификатором транзакции.

Следует помнить, что фиксация транзакции освобождает любые блокировки строк и таблиц, установленные вашим сеансом, например, для команды SELECT FOR UPDATE. Кроме того, удаляются все точки сохранения, установленные после последней операции COMMIT или ROLLBACK.

После того как изменения зафиксированы (COMMIT), их уже невозможно отменить при помощи оператора ROLLBACK.

Приведем несколько примеров корректного использования оператора COMMIT:

```
COMMIT;
COMMIT WORK;
COMMIT COMMENT 'maintaining account balance'.
```

Оператор ROLLBACK

При выполнении оператора ROLLBACK отменяются все или некоторые изменения, внесенные вашим сеансом в базу данных в рамках текущей транзакции. Почему может возникнуть желание отменить изменения? Что касается SQL, оператор ROLLBACK обеспечивает возможность исправления возможных ошибок, например:

```
DELETE FROM orders;
```

«О нет, я хотел удалить только те заказы, которые были сделаны до мая 1995 года!!!» Нет проблем – просто выполните ROLLBACK. С точки зрения кодирования приложения смысл ROLLBACK в том, что он позволяет при возникновении проблемы вернуться в исходное состояние – начать все «с чистого листа».

Синтаксис оператора ROLLBACK:

```
ROLLBACK [WORK] [TO [SAVEPOINT] имя_точки_сохранения];
```

Можно использовать оператор ROLLBACK в одном из двух режимов: без указания параметров или с предложением TO, указывающим точку сохранения, до которой следует откатить изменения. ROLLBACK без параметров отменяет все изменения, внесенные в рамках транзакции.

Форма ROLLBACK TO позволяет отменить все изменения и освободить все блокировки, сделанные после того, как была установлена точка сохранения с меткой *имя_точки_сохранения* (об установке точек сохранения будет подробно рассказано в следующем разделе[1], посвященном оператору SAVEPOINT).

Параметр *имя_точки_сохранения* – это необъявляемый идентификатор Oracle, который не может быть ни литералом (заключенным в кавычки), ни именем переменной.

Допустимы все нижеперечисленные примеры использования ROLLBACK:

```
ROLLBACK;
ROLLBACK WORK;
ROLLBACK TO begin_cleanup;
```

При выполнении отката до определенной точки сохранения все точки сохранения, установленные после точки с меткой *имя_точки_сохранения*, удаляются, но сама эта точка остается. То есть вы сможете возобновить свою транзакцию с данной точки и при необходимости снова откатить изменения до этой точки при возникновении другой ошибки.

PL/SQL неявно генерирует точку сохранения непосредственно перед выполнением оператора INSERT, UPDATE или DELETE. И в случае неудачного исполнения оператора DML откат изменений автоматически осуществляется до данной неявной точки сохранения. Таким образом, отменяется только последний оператор DML.

Автономные транзакции

Определяя PL/SQL-блок (анонимный блок, процедуру, функцию, пакетную процедуру, пакетную функцию, триггер базы данных) как *автономную транзакцию*, вы изолируете в этом блоке команды DML от контекста транзакции вызывающего приложения. Такой блок становится независимой транзакцией, которая запускается другой транзакцией, называемой *главной* по отношению к данной.

Во время выполнения блока автономной транзакции главная транзакция приостанавливается. Вы выполняете SQL-операции, фиксируете их или откатываете, затем возобновляете главную транзакцию (рис. 1.5).

---

[1] Снова ошибка в оригинале книги. Такого раздела нет. – *Примеч. перев.*

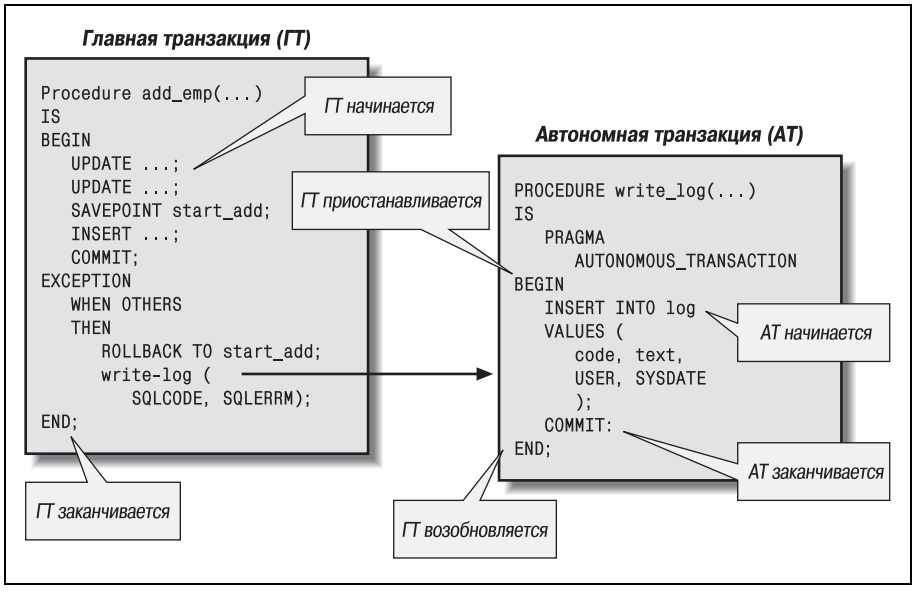


Рис. 1.5. *Передача управления между главной и автономной транзакциями*

Определить PL/SQL-блок как автономную транзакцию несложно: в раздел объявлений включается директива:

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

Данная директива указывает компилятору PL/SQL, что следует определить блок PL/SQL как автономный (независимый). Следующие виды PL/SQL-блоков могут быть определены как автономная транзакция:

- Анонимные блоки PL/SQL верхнего уровня (но не вложенные)
- Функции и процедуры, определенные внутри пакета или являющиеся самостоятельными программами
- Методы (функции и процедуры) объектного типа
- Триггеры базы данных

Директива автономной транзакции может размещаться в любой части раздела объявлений конкретного PL/SQL-блока. Однако, вероятно, лучше всего поместить ее перед объявлениями всех структур данных. В этом случае любой программист, читающий ваш код, сразу же определит, что программа представляет собой автономную транзакцию.

Данная директива – единственное добавление в синтаксис языка, потребовавшееся для поддержки автономных транзакций в PL/SQL. `COMMIT`, `ROLLBACK`, команды DML использовались и ранее. Однако при исполнении внутри автономной транзакции эти операторы имеют другую область действия и видимости; необходимо явно включить `COMMIT` или `ROLLBACK` в свою программу.

Триггеры базы данных

Триггеры базы данных – это именованные элементы программы, которые исполняются в ответ на наступление каких-то событий в базе данных. Триггеры могут быть связаны с четырьмя различными видами событий:

Операторы DML

Триггеры DML могут запускаться при вставке, обновлении или удалении записи таблицы. Эти триггеры могут использоваться для проверки данных, установки значений по умолчанию, отслеживания изменений и даже запрещения некоторых DML-операций.

Операторы DDL

Триггеры DDL запускаются при выполнении команд DDL, например при создании таблицы. Они могут обеспечивать отслеживание выполняемых действий и запрещать выполнение некоторых команд DDL.

События базы данных

Триггеры событий базы данных запускаются при запуске и остановке базы данных, при подключении и отключении пользователя или при возникновении ошибки Oracle. Начиная с версии Oracle8*i* Database эти триггеры позволяют отслеживать активность в базе данных.

`INSTEAD OF`

Триггеры `INSTEAD OF` (замещающие триггеры) являются, по сути, альтернативой триггерам DML. Они запускаются перед выполнением вставки, обновления или удаления, при этом их код указывает, что следует делать вместо этих DML-операций. Триггеры `INSTEAD OF` управляют операциями над представлениями, а не над таблицами. Они могут использоваться для того, чтобы сделать необновляемое представление обновляемым и для того, чтобы изменить поведение обновляемых представлений. Далее мы не будем говорить об этих триггерах, так как это особая тема, требующая серьезного изучения.

Триггеры DML

Триггеры языка манипулирования данными (Data Manipulation Language – DML) запускаются при вставке, обновлении или удалении строк в определенной таблице. Это самый распространенный тип триггеров, особенно среди разработчиков (остальными типами триггеров в основном пользуются администраторы баз данных).

Существует множество вариантов использования триггеров DML. Они могут запускаться до или после выполнения оператора DML, а также до или после обработки каждой строки внутри команды. Триггеры DML могут запускаться при выполнении операторов `INSERT`, `UPDATE`, `DELETE`, а также их комбинаций.

Участие в транзакциях

По умолчанию триггеры DML принимают участие в транзакциях, из которых они запускаются, то есть:

- Если триггер инициирует исключение, то соответствующая часть транзакции будет отменена.
- Если триггер сам выполняет какие-то операторы DML (например, вставляет запись в журнальную таблицу), то такие операторы DML становятся частью главной транзакции.
- Внутри триггера DML нельзя использовать операторы COMMIT и ROLLBACK.

Однако если вы определяете триггер DML как автономную транзакцию, то любые команды DML, исполняемые внутри триггера, будут сохранены или отменены посредством явно использованного оператора COMMIT или ROLLBACK, при этом главная транзакция затрагиваться не будет.

В последующих разделах будет описано создание триггера DML, описаны различные элементы определения триггера, а также рассмотрен пример, использующий многие компоненты и возможности триггеров DML.

Создание триггера DML

Используйте для создания (или пересоздания) триггера DML следующую конструкцию:

```
1  CREATE [OR REPLACE] TRIGGER имя_триггера
2  {BEFORE | AFTER}
3  {INSERT | DELETE | UPDATE | UPDATE OF список_столбцов} ON имя_таблицы
4  [FOR EACH ROW]
5  [WHEN (...)]
6  [DECLARE ... ]
7  BEGIN
8    ... исполняемые операторы ...
9  [EXCEPTION ... ]
10 END [имя-триггера];
```

Описание элементов объявления приведено в таблице:

Строки	Описание
1	Сообщает о том, что создается триггер с указанным именем. Предложение OR REPLACE является необязательным. Если триггер уже существует, а REPLACE отсутствует, то попытка создать триггер заново приведет к возникновению ошибки ORA-4081. Совпадение имен, например таблицы и триггера (или процедуры и триггера), теоретически разрешено, но мы рекомендуем воздержаться от таких совпадений во избежание путаницы.
2	Указывает, должен ли триггер запускаться до (BEFORE) или после (AFTER) того, как обработана команда или строка.

Строки	Описание
3	Указывает, к какому оператору DML будет применяться триггер: INSERT, UPDATE или DELETE. Обратите внимание, что UPDATE может использоваться как для целой записи, так и для разделенного запятыми списка столбцов. Столбцы можно комбинировать (при помощи OR) и указывать в любом порядке. В строке 3 также указана таблица, с которой работает триггер. Имейте в виду, что любой триггер DML может обрабатывать только одну таблицу.
4	Наличие предложения FOR EACH ROW означает, что триггер будет запускаться для каждой строки, обрабатываемой командой. Если данное предложение отсутствует, то по умолчанию триггер запускается только один раз для данной команды (триггер уровня команды).
5	Необязательное предложение WHEN, которое позволяет определить логическое условие, позволяющее избежать ненужного исполнения триггера.
6	Необязательный раздел объявлений анонимного блока, составляющего код триггера. Если вам не нужно объявлять локальные переменные, то в этом ключевом слове нет необходимости. Имейте в виду, что ни при каких условиях не следует объявлять псевдозаписи NEW и OLD: это делается автоматически.
7–8	Исполняемый раздел триггера. Является обязательным и должен включать в себя хотя бы один оператор.
9	Необязательный раздел исключений. Здесь будут перехватываться и обрабатываться любые исключения, порождаемые в исполняемом разделе.
10	Обязательный оператор END для триггера. После ключевого слова END можно для наглядности указать имя заканчивающегося триггера.

Рассмотрим несколько примеров использования триггеров DML:

- Хотелось бы иметь уверенность в том, что при добавлении данных нового сотрудника или изменении информации о сотруднике выполняются все необходимые проверки. Обратите внимание, что в этом триггере уровня строк необходимые поля псевдозаписи NEW передаются отдельным программам проверки:

```
CREATE OR REPLACE TRIGGER validate_employee_changes
   AFTER INSERT OR UPDATE
   ON employee
   FOR EACH ROW
BEGIN
   check_age (:NEW.date_of_birth);
   check_resume (:NEW.resume);
END;
```

- Следующий триггер BEFORE INSERT осуществляет аудит для таблицы ceo_compensation. Для сохранения новой строки без изменения «внешней» (главной) транзакции триггер полагается на функциональность автономных транзакций Oracle8*i* Database:

```
CREATE OR REPLACE TRIGGER bef_ins_ceo_comp
   AFTER INSERT
   ON ceo_compensation
   FOR EACH ROW
DECLARE
   PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
   INSERT INTO ceo_comp_history
        VALUES (:NEW.name,
                :OLD.compensation, :NEW.compensation,
                'AFTER INSERT', SYSDATE);
   COMMIT;
END;
```

Предложение WHEN

Используйте предложение WHEN для уточнения условий выполнения кода триггера. В следующем примере предложение WHEN позволяет обеспечить исполнение кода триггера только при *изменении* величины заработной платы (salary):

```
CREATE OR REPLACE TRIGGER check_raise
   AFTER UPDATE OF salary
   ON employee
   FOR EACH ROW
WHEN  (OLD.salary != NEW.salary) OR
      (OLD.salary IS NULL AND NEW.salary IS NOT NULL) OR
      (OLD.salary IS NOT NULL AND NEW.salary IS NULL)
BEGIN
   ...
```

Другими словами, если пользователь выполняет команду UPDATE для строки, и по какой-то причине значение зарплаты остается неизменным, то триггер должен запуститься, и он запустится, но ведь в действительности исполнение PL/SQL-кода тела триггера при этом не требуется. Если вы укажете соответствующее условие в предложении WHEN, вам удастся избежать расходов, связанных с ненужным исполнением PL/SQL-блока триггера.

Файл *genwhen.sp* на веб-сайте этой книги содержит процедуру, которая формирует предложение WHEN для проверки отличия нового значения от старого.

В большинстве случаев предложение WHEN ссылается на поля псевдозаписей OLD и NEW, как и в рассмотренном примере. Однако можно написать код, который будет вызывать встроенные функции. В следующем примере предложение WHEN использует SYSDATE для того, чтобы ограничить время исполнения триггера INSERT периодом с 9 часов утра до 5 часов вечера:

```
CREATE OR REPLACE TRIGGER valid_when_clause
BEFORE INSERT ON frame
```

```
FOR EACH ROW
WHEN ( TO_CHAR(SYSDATE,'HH24') BETWEEN 9 AND 17 )
   ...
```

Псевдозаписи NEW и OLD

При запуске триггера уровня строки исполняющее ядро PL/SQL создает и заполняет две структуры данных, которые работают подобно записям. Это псевдозаписи NEW и OLD (приставка «псевдо» объясняется тем, что они обладают не всеми свойствами настоящих записей PL/SQL). OLD хранит начальные значения записи, обрабатываемой триггером, а NEW – новые значения. Эти записи имеют такую же структуру, как запись, объявленная при помощи атрибута %ROWTYPE на основе таблицы, к которой относится триггер.

При работе с псевдозаписями NEW и OLD необходимо учитывать несколько правил:

- Для триггеров, относящихся к командам INSERT, структура OLD не содержит никаких данных, «старого» набора значений *не существует*.
- Для триггеров, относящихся к командам UPDATE, заполняются обе структуры: OLD и NEW. OLD содержит исходные значения (до обновления), а NEW – те значения, которые получит строка после выполнения обновления.
- Для триггеров, относящихся к командам DELETE, структура NEW не содержит никаких данных, ведь речь идет об удалении записи.
- Изменение значений полей структуры OLD запрещено, попытка такого изменения приведет к возникновению ошибки ORA-04085. Изменение значений полей структуры NEW *допустимо*.
- Структура NEW или OLD не может передаваться как параметр типа запись в процедуру или функцию, вызываемую внутри триггера. Можно передавать только отдельные поля псевдозаписей. В файле *gentrigrec.sp* на веб-сайте этой книге приведена программа, формирующая код, который передает значения NEW и OLD записям, которые уже *могут* передаваться как параметры.
- При ссылке на структуру NEW или OLD внутри анонимного блока триггера[1] необходимо предварять соответствующие ключевые слова двоеточием, например:

```
IF :NEW.salary > 10000 THEN...
```

- Выполнение операций уровня записи для структур NEW и OLD не поддерживается. Например, подобный код вызовет ошибку при компиляции триггера:

```
BEGIN :new := NULL; END;
```

[1] Очевидно, имеется в виду секция исполнения триггера. – *Примеч. науч. ред.*

Определение DML-действия внутри триггера

Oracle предлагает набор функций (называемых также *операционными директивами*), которые позволяют определить, какое DML-действие вызвало запуск текущего триггера. Каждая такая функция возвращает TRUE или FALSE.

INSERTING

Возвращает TRUE, если триггер был запущен в ответ на вставку в таблицу, с которой связан триггер, и FALSE – в противном случае.

UPDATING

Возвращает TRUE, если триггер был запущен в ответ на обновление таблицы, с которой связан триггер, и FALSE – в противном случае.

DELETING

Возвращает TRUE, если триггер был запущен в ответ на удаление из таблицы, с которой связан триггер, и FALSE – в противном случае.

Используя эти директивы, можно создать один общий триггер, который будет объединять действия, необходимые для различных видов операций.

Триггеры DDL

Oracle позволяет определить триггеры, которые будут запускаться в ответ на исполнение операторов языка DDL (Data Definition Language – язык определения данных). Попросту говоря, оператор DDL – это любой оператор SQL, используемый для создания или изменения объекта базы данных, такого как таблица или индекс. Приведем несколько примеров операторов DDL:

- CREATE TABLE
- ALTER INDEX
- DROP TRIGGER

Каждый из этих операторов приводит к созданию, изменению или удалению объекта базы данных.

Синтаксис создания таких триггеров практически совпадает с синтаксисом создания триггеров DML, отличие лишь в запускающих их событиях и в том, что триггеры DDL не применяются к отдельным таблицам.

Создание триггера DDL

Для создания (или пересоздания) триггера DDL используйте такую конструкцию:

```
1  CREATE [OR REPLACE] TRIGGER имя_триггера
2  {BEFORE | AFTER } {DDL-событие} ON {DATABASE | SCHEMA}
3  [WHEN (...)]
4  DECLARE
```

```
5  Объявления переменных
6  BEGIN
7  ... код...
8  END;
```

Описание элементов объявления приведено в таблице:

Строки	Описание
1	Сообщает о том, что создается триггер с указанным именем. Предложение OR REPLACE является необязательным. Но если триггер уже существует, а REPLACE отсутствует, то попытка создать триггер заново приведет к возникновению старой доброй ошибки ORA-4081.
2	Это очень важная строка. Она определяет, должен ли триггер запускаться до, после или вместо определенного DDL-события*, а также должен ли он запускаться для всех операций над базой данных или только в рамках текущей схемы. Имейте в виду, что опция INSTEAD OF доступна только в версии Oracle9*i* Release 1 и выше.
3	Необязательное предложение WHEN, которое позволяет определить логику, позволяющую избежать ненужного исполнения триггера.
4–7	В этих строках приводится PL/SQL-содержимое триггера.

* INSTEAD OF недопустимо в триггерах DDL. – *Примеч. науч. ред.*

Рассмотрим пример триггера, выполняющего роль необученного информатора-«глашатая», объявляющего о создании всех объектов:

```
/* Файл на веб-сайте: uninformed_town_crier.sql */
SQL> CREATE OR REPLACE TRIGGER town_crier
  2  AFTER CREATE ON SCHEMA
  3  BEGIN
  4    DBMS_OUTPUT.PUT_LINE('I believe you have created something!');
  5  END;
  6  /
Trigger created.
```

Триггеры событий базы данных

Триггеры событий базы данных запускаются при возникновении событий на уровне базы данных. Существует пять триггеров событий базы данных:

STARTUP

Запускается при запуске базы данных.

SHUTDOWN

Запускается при нормальной остановке базы данных.

SERVERERROR

Запускается при возникновении ошибки Oracle.

LOGON

Запускается при открытии сеанса Oracle.

LOGOFF

Запускается при нормальном завершении сеанса Oracle.

Администраторы базы данных сразу обратят внимание на то, что эти триггеры представляют собой великолепное средство автоматизации процесса администрирования базы данных и обеспечения детального контроля над базой данных.

Создание триггера события базы данных

Синтаксис, используемый для создания такого триггера, очень похож на синтаксис создания триггера DDL:

```
1  CREATE [OR REPLACE] TRIGGER имя_триггера
2  {BEFORE | AFTER} {событие_базы_данных} ON {DATABASE | SCHEMA}
3  DECLARE
4  Объявление переменных
5  BEGIN
6  ... код...
7  END;
```

Существует ряд ограничений, накладываемых на использование атрибутов BEFORE и AFTER для определенных событий. Некоторые ситуации представляются просто бессмысленными:

Не бывает триггеров BEFORE STARTUP

Даже если бы такой триггер и можно было бы создать, когда бы он запускался? Попытка создания триггера такого вида приводит к появлению очевидного сообщения об ошибке:

```
ORA-30500: database open triggers and server error triggers
                                          cannot have BEFORE type
```

Не бывает триггеров AFTER SHUTDOWN

Опять-таки, когда бы такой триггер мог запускаться? Попытки создания такого триггера отвергаются с выдачей следующего сообщения об ошибке:

```
ORA-30501: instance shutdown triggers cannot have AFTER type
```

Не бывает триггеров BEFORE LOGON

Для реализации таких триггеров потребовался бы какой-то чрезвычайно проницательный код: «Слушай, мне кажется, кто-то собирается подключиться: давай-ка сделаем что-нибудь!». Будучи реалистом, Oracle не позволяет создавать такие триггеры, выдавая сообщение об ошибке:

```
ORA-30508: client logon triggers cannot have BEFORE type
```

Не бывает триггеров AFTER LOGOFF

«Нет, пожалуйста, вернись! Только не отключайся!»... Попытки создания этих триггеров завершаются появлением следующего сообщения:

```
ORA-30509: client logoff triggers cannot have AFTER type
```

Не бывает триггеров BEFORE SERVERERROR

Подобный триггер был бы мечтой каждого программиста! Только представьте себе, что было бы возможно такое:

```
CREATE OR REPLACE TRIGGER BEFORE_SERVERERROR
BEFORE SERVERERROR ON DATABASE
BEGIN
  diagnose_impending_error;
  fix_error_condition;
  continue_as_if_nothing_happened;
END;
```

Но, к сожалению, наши мечты прерывает сообщение:

```
ORA-30500: database open triggers and server error triggers
                                          cannot have BEFORE type
```

Динамический SQL и динамический PL/SQL

Динамический SQL подразумевает под собой те операторы SQL, которые формируются и исполняются во время исполнения программы. Термин «динамический» – антоним для термина «статический». *Статический SQL* – это операторы SQL, которые фиксируются в момент компиляции программы и далее не изменяются. *Динамический PL/SQL* соответственно понимается как целые PL/SQL-блоки кода, которые динамически формируются, затем компилируются и исполняются.

С выходом версии Oracle7 Release 1 разработчики PL/SQL получили возможность использовать для исполнения динамического SQL встроенный пакет DBMS_SQL. В Oracle8*i* Database появилась дополнительная возможность исполнения динамически формируемых команд SQL – NDS (native dynamic SQL – *встроенный динамический SQL*). NDS входит в состав языка PL/SQL; его гораздо легче использовать, чем DBMS_SQL, к тому же во многих приложениях он исполняется более эффективно.

Оператор EXECUTE IMMEDIATE

Используйте оператор EXECUTE IMMEDIATE для (немедленного!) исполнения указанной команды SQL:

```
EXECUTE IMMEDIATE строка_SQL
   [INTO {переменная[, переменная]... | запись}]
   [USING [IN | OUT | IN OUT] аргумент_связывания
       [, [IN | OUT | IN OUT] аргумент_связывания]...];
```

где:

строка_SQL

Строковое выражение, содержащее команду SQL или PL/SQL-блок.

переменная

Переменная, которая получает значение столбца, возвращенное запросом.

запись

Запись, основанная на определенном пользователем типе или атрибуте `%ROWTYPE`, которая получает целую строку, возвращенную запросом.

аргумент_связывания

Выражение, значение которого передается в команду SQL или PL/SQL-блок, или идентификатор, который служит входной и/или выходной переменной для функции или процедуры, вызываемой в PL/SQL-блоке.

`INTO` ***предложение***

Используется для однострочных запросов. Для каждого значения столбца, возвращенного запросом, следует указать отдельную переменную или поле записи совместимого типа.

`USING` ***предложение***

Используется для передачи аргументов связывания в строку SQL. Это предложение используется как для динамического SQL, так и для динамического PL/SQL, поэтому существует возможность указать режим передачи параметров (имеет смысл только для PL/SQL; по умолчанию установлен в значение «IN», соответствующее единственному виду аргументов, доступных для команд SQL).

Оператор `EXECUTE IMMEDIATE` может использоваться для любой SQL-команды или PL/SQL-блока за исключением многострочных запросов. Если *строка_SQL* заканчивается точкой с запятой, то она воспринимается как PL/SQL-блок. В противном случае она воспринимается как `SELECT`, оператор DML (`INSERT`, `UPDATE` или `DELETE`) или DDL (например, `CREATE TABLE`). В строке могут присутствовать заполнители для аргументов связывания, но с их помощью нельзя передавать имена объектов схемы, такие как имена таблиц и столбцов.

Если в вашей программе выполняется оператор DDL, то вместе с ним выполнится и фиксация изменений. Если вы не хотите, чтобы фиксация изменений, вызванная оператором DDL, повлияла на ранее сделанные изменения в остальной части приложения, то следует поместить динамический оператор DDL в процедуру, реализованную как автономная транзакция (см. пример в файле *auton_ddl.sql* на веб-сайте этой книги).

При выполнении команды исполняющее ядро заменяет каждый заполнитель (идентификатор, перед которым стоит двоеточие, например `:salary_value`) в *строке_SQL* соответствующим *аргументом_связывания* (в соответствии с позицией). Вы можете передавать числа, даты и строки.

Логические выражения передавать нельзя, так как BOOLEAN — это тип данных PL/SQL. Также нельзя передавать литеральное значение NULL (вместо этого следует передавать переменную разрешенного типа, имеющую значение NULL).

NDS поддерживает все типы данных SQL. *Переменные* и *аргументы_связывания* могут быть коллекциями, большими объектами (типа LOB), экземплярами объектного типа и типа REF. Типы данных, специфичные для PL/SQL, NDS не поддерживает: тип BOOLEAN, ассоциативные массивы и пользовательские типы записей. При этом *предложение* INTO может содержать PL/SQL-запись.

Давайте рассмотрим несколько примеров:

- Создание индекса:

```
EXECUTE IMMEDIATE 'CREATE INDEX emp_u_1 ON employee (last_name)';
```

  PL/SQL не поддерживает встроенные команды DDL; необходимо использовать динамический SQL.

- Получение количества строк некоторой таблицы в некоторой схеме для указанного предложения WHERE:

```
/* Файл на веб-сайте:_nds.sf */
CREATE OR REPLACE FUNCTION tabcount (
   tab IN VARCHAR2, whr IN VARCHAR2 := NULL)
   RETURN PLS_INTEGER AUTHID CURRENT_USER
IS
   str      VARCHAR2 (32767) := 'SELECT COUNT(*) FROM ' || tab;
   retval   PLS_INTEGER;
BEGIN
   IF whr IS NOT NULL
   THEN
      str := str || ' WHERE ' || whr;
   END IF;

   EXECUTE IMMEDIATE str INTO retval;
   RETURN retval;
EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.put_line (
         'TABCOUNT ERROR: ' || DBMS_UTILITY.FORMAT_ERROR_STACK);
      DBMS_OUTPUT.put_line (str);
      RETURN NULL;
END;
/
```

  Теперь уже можно больше не писать SELECT COUNT(*) ни в SQL*Plus, ни в программе на PL/SQL. Для подсчета количества строк поступаем следующим образом:

```
BEGIN
   IF tabCount ('emp', 'deptno = ' || v_dept) > 100
```

```
   THEN
      DBMS_OUTPUT.PUT_LINE ('Growing fast!');
   END IF;
```

- Функция, позволяющая обновить значение любого числового столбца в таблице employee. Реализовано в виде функции, чтобы можно было дополнительно возвращать количество обновленных строк.

```
/* Файл на веб-сайте: updnval.sf */
CREATE OR REPLACE FUNCTION updNVal (
   col IN VARCHAR2,
   val IN NUMBER,
   start_in IN DATE,
   end_in IN DATE)
   RETURN PLS_INTEGER
IS
BEGIN
   EXECUTE IMMEDIATE
      'UPDATE employee SET ' || col || ' = :the_value
        WHERE hire_date BETWEEN :lo AND :hi'
     USING val, start_in, end_in;
   RETURN SQL%ROWCOUNT;
END;
```

И для достижения подобной гибкости требуется совсем небольшой объем кода! В примере использован аргумент связывания: после синтаксического анализа команды UPDATE ядро PL/SQL заменяет заполнители :the_value, :lo и :hi значениями из предложения USING. Обратите внимание, что, как и в случае со статическими командами SQL, можно использовать атрибут курсора SQL%ROWCOUNT.

Как видите, синтаксис оператора EXECUTE IMMEDIATE очень прост и доступен!

Оператор OPEN FOR

PL/SQL-оператор OPEN FOR *не* создавался специально для работы с NDS; он появился в версии Oracle7 и обеспечивал поддержку курсорных переменных. Теперь этот оператор используется для элегантной реализации многострочных динамических запросов. Использование DBMS_SQL для выполнения многострочных запросов представляло собой мучительную многоэтапную процедуру: синтаксический анализ, связывание, определение каждого столбца в отдельности, выборка, извлечение значений каждого столбца в отдельности. Приходилось писать значительный объем кода!

Разработчики Oracle использовали существующую функциональность – курсорные переменные – и сохранили синтаксис, весьма естественно расширив его для поддержки динамического SQL. Давайте посмотрим на синтаксис оператора OPEN FOR:

```
OPEN {курсорная_переменная | :внешняя_курсорная_переменная} FOR SQL_строка
```

```
[USING аргумент_связывания[, аргумент_связывания]...];
```

где:

курсорная_переменная

Слабо типизированная курсорная переменная.

:внешняя_курсорная_переменная

Курсорная переменная, объявленная в среде, вызывающей PL/SQL (например, в программе Oracle Call Interface – OCI).

SQL_строка

Содержит команду SELECT, подлежащую динамическому исполнению.

USING *предложение*

Следует тем же правилам, что и для оператора EXECUTE IMMEDIATE.

Если вы работаете с версией Oracle9*i* Database Release 2 или Oracle Database 10*g*, то можете использовать EXECUTE IMMEDIATE в сочетании с BULK COLLECT для извлечения нескольких строк в динамическом запросе. Такой подход требует гораздо меньшего объема кода и может значительно улучшить производительность работы вашего запроса.

Рассмотрим пример, в котором объявляется слабо типизированный REF CURSOR, курсорная переменная на основе этого типа, а затем при помощи оператора OPEN FOR открывается динамический запрос:

```
CREATE OR REPLACE PROCEDURE show_parts_inventory (
   parts_table IN VARCHAR2,
   where_in IN VARCHAR2)
IS
   TYPE query_curtype IS REF CURSOR;
   dyncur query_curtype;
BEGIN
   OPEN dyncur FOR
      'SELECT * FROM ' || parts_table ||
      ' WHERE ' || where_in;
   ...
```

После того как запрос открыт, используем для извлечения строк, закрытия курсорной переменной и проверки атрибутов курсора такой же синтаксис, как для статических курсорных переменных и жестко закодированных явных курсоров.

Динамический PL/SQL

Динамический PL/SQL предоставляет ряд интереснейших перспективных возможностей по разработке кода. Только подумайте: в то время как пользователь работает с вашим приложением, вы (с помощью NDS) можете выполнять следующие действия:

- Создавать программу, в том числе и пакет, содержащий глобальные структуры данных.
- Получать (и изменять) значения глобальных переменных по именам.
- Вызывать функции и процедуры, чьи имена не известны на момент компиляции.

При работе с динамическими блоками PL/SQL и NDS следует помнить следующее:

- Динамическая строка должна являться корректным PL/SQL-блоком. Она должна начинаться с ключевого слова `DECLARE` или `BEGIN` и заканчиваться оператором `END` и точкой с запятой. В отсутствие завершающей точки с запятой строка не будет восприниматься как PL/SQL-код.
- В динамическом блоке доступны только глобальные элементы PL/SQL (отдельные функции и процедуры и элементы, определенные в спецификации пакета). Динамические блоки PL/SQL исполняются вне локального внешнего блока.
- Ошибки, возникающие в динамическом блоке PL/SQL, могут перехватываться и обрабатываться локальным блоком, в котором оператором `EXECUTE IMMEDIATE` была запущена данная строка.

Рассмотрим пример использования динамического PL/SQL. Речь пойдет о реальной истории. Я работал консультантом в страховой компании в Чикаго, и меня попросили посмотреть, можно ли что-то сделать с одной программой, создающей множество проблем. Она была очень большой, и размер ее продолжал расти – скоро можно было ожидать трудностей уже на этапе компиляции. К моему удивлению оказалось, что эта программа выглядит следующим образом:

```
CREATE OR REPLACE PROCEDURE process_line (line IN INTEGER)
IS
BEGIN
   IF    line = 1 THEN process_line1;
   ELSIF line = 2 THEN process_line2;
   ...
   ELSIF line = 514 THEN process_line514;
   ...
   ELSIF line = 2057 THEN process_line2057;
   END IF;
END;
```

Каждому значению номера строки (`line`) соответствовала специальная программа `process_line`, которая обрабатывала указанные условия. По мере добавления страховой компанией новых условий в свой договор обслуживания программа все росла и росла. Абсолютно немасштабируемый подход!

Чтобы избежать такого беспорядка, программист должен найти повторяющиеся элементы кода. Выявив такое повторение, следует создать программу для многократного использования и заключить в нее повторяющийся код (шаблон) или попытаться записать его с помощью конструкции динамического SQL.

Мне удалось заменить эти тысячи строк кода следующим фрагментом:

```
CREATE OR REPLACE PROCEDURE process_line (line IN INTEGER)
IS
BEGIN
   EXECUTE IMMEDIATE
      'BEGIN process_line' || line || '; END;';
END;
```

Тысячи строк превратились в один исполняемый оператор! Конечно, далеко не всегда поиск шаблона и его преобразование в динамический SQL будет таким очевидным. Но потенциальная выгода просто огромна!

Заключение: от основ к применению PL/SQL

Прочитав эту главу, вы получили достаточно серьезные базовые знания по языку PL/SQL. Познакомившись с основами (и, я надеюсь, имея некоторый опыт по созданию реальных программ PL/SQL), вы теперь можете переходить к главам, в которых описана специфическая функциональность PL/SQL, необходимая для выполнения работы администратора баз данных.

2

Курсоры

Курсор – это средство извлечения данных из базы данных Oracle. Курсоры содержат определения столбцов и объектов (таблиц, представлений и т. п.), из которых будут извлекаться данные, а также набор критериев, определяющих, какие именно строки должны быть выбраны. Для начала дадим два примера очень простых курсоров:

```
SELECT name
  FROM emp;

CURSOR curs_get_emp IS
SELECT name
  FROM emp;
```

Администраторы Oracle, конечно, знакомы с основным синтаксисом курсоров, но для описания их функциональности (особенно в первом примере) они могут использовать термины «оператор SELECT», «запрос» или «выборка». В целом, это именно то, что делают курсоры, – извлекают данные. Большинство администраторов знают, что курсоры глубоко интегрированы в механизмы базы данных, но не всем известно, что понимание устройства курсоров и управления ими может радикально улучшить производительность приложений для СУБД Oracle.

Предназначенные изначально для выборки данных, курсоры стали неотъемлемой частью PL/SQL. В этой главе рассматривается взаимодействие курсоров с PL/SQL с точки зрения администратора базы данных. В ней рассказывается, как повторное использование курсоров может повысить производительность, обсуждаются различия между явными и неявными курсорами и то, как каждый из них может влиять на вашу базу данных. В ней также исследуется выигрыш в производительности от оптимизации мягко закрываемых курсоров в Oracle. Кроме того, в этой главе обсуждаются курсоры типа `REF` (динамические), передача курсоров в качестве параметров и курсорные выражения (вложенные курсоры).

На протяжении нескольких лет шла оживленная дискуссия о правильном использовании курсоров в Oracle. В частности, администраторы и разработчики спорили, какой из типов курсоров работает быстрее, или задавались вопросом об эффективности модели повторного использования в Oracle. Прошли годы, и в последних версиях Oracle курсоры были значительно усовершенствованы. Эта глава не имеет целью склонить вас к определенному способу реализации и настройки курсоров в вашей системе. В значительной мере способ использования курсоров определяется характеристиками вашей организации, используемыми в ней данными и приложениями. Здесь мы пытаемся объяснить основные возможности и дать некоторые рекомендации, которые смогут помочь вам сделать правильный выбор.

Повторное использование курсоров

В основе концепции повторного использования курсоров лежит очень простая идея: после использования курсор может быть использован еще раз. Говоря точнее, скомпилированная версия курсора может использоваться повторно во избежание расходов на разбор и повторную компиляцию.

Полный и частичный разбор

Процесс компиляции нового курсора называется *полным разбором* (и сам по себе заслуживает отдельной книги); в контексте этой главы он может быть упрощенно представлен четырьмя этапами:

Проверка

Курсор проверяется на соответствие синтаксическим правилам SQL, также проверяются объекты (таблицы и столбцы), на которые он ссылается.

Компиляция

Курсор компилируется в исполняемый вид и загружается в разделяемый пул сервера базы данных. Для определения местоположения курсора в разделяемом пуле используется его *адрес*.

Вычисление плана выполнения

Оптимизатор по стоимости (cost-based optimizer – CBO) Oracle определяет наилучший для данного курсора план выполнения и присоединяет его к курсору.

Вычисление хеша

ASCII-значения всех символов курсора складываются и передаются в функцию хеширования. Эта функция рассчитывает значение, по которому курсор легко может быть найден при повторном обращении. Данное значение называется *хеш-значением* курсора. Ниже в этом разделе мы еще вернемся к ASCII-значениям.

Значительная часть активности защелок БД приходится на период выполнения этих операций, так как Oracle не может позволить изменять используемые курсором объекты (таблицы и столбцы) во время его проверки и компиляции. Выполнение этих задач почти полностью ложится на процессор, поэтому во время компиляции наблюдается большой расход процессорного времени сервера БД. Здесь важно то, что основная работа по извлечению записей из-за этого откладывается. Последовательное выполнение одного и того же курсора (одной или несколькими программами) позволяет заменить дорогостоящий процесс полного разбора простой проверкой наличия доступа к используемым объектам (таблицам, представлениям и т. п.) и перейти сразу к извлечению записей. Благодаря этому экономится значительное время.

Общее количество выполненных базой данных полных разборов можно получить из таблицы V$SYSSTAT следующим запросом:

```
SQL> SELECT name,
  2         value
  3    FROM v$sysstat
  4   WHERE name = 'parse count (hard)';

NAME                                VALUE
------------------------------ ----------
parse count (hard)                     676
```

Если количество полных разборов в некотором приложении постоянно растет, то это говорит о том, что оно не использует в полной мере преимущества повторного использования курсоров.

Даже после того, как курсор подвергся полному разбору, при повторном использовании он может потребовать дополнительного разбора. Однако этот процесс будет гораздо менее затратным, он в основном сводится к проверке безопасности, удостоверяющей, что запросивший выполнение курсора пользователь имеет права на доступ к его объектам. Такая неполная обработка называется *частичным разбором*.

Общее количество выполненных базой данных частичных разборов можно рассчитать, вычтя из общего числа разборов число полных разборов:

```
SQL> SELECT ( SELECT value
  2             FROM v$sysstat
  3            WHERE name = 'parse count (total)' )
  4       - ( SELECT value
  5             FROM v$sysstat
  6            WHERE name = 'parse count (hard)' ) soft_parse
  7    FROM dual;

SOFT_PARSE
----------
      4439
```

Время, затраченное базой данных на разбор (процессорное и фактическое), тоже можно получить из таблицы V$SYSSTAT.

```
SQL> SELECT name,
  2         value
  3    FROM v$sysstat
  4  WHERE name LIKE 'parse time%';

NAME                                VALUE
------------------------------ ----------
parse time cpu                         381
parse time elapsed                    5933
```

В идеальном случае эти значения не должны заметно расти при работе приложения. Однако в действительности некоторое увеличение почти всегда происходит, так как даже простейший частичный разбор потребляет *некоторое количество* процессорного времени.

Планирование использования курсора

Хорошей практикой является ограничение количества разборов курсора – оптимальное значение равно, конечно, единице. Одним из путей в достижении идеала будет предварительный разбор всех курсоров, которые, возможно, будут выполняться вашим приложением. В таком случае при старте приложения все курсоры уже будут ждать его в разделяемом пуле. Однако этот подход связан с большой трудоемкостью при сопровождении больших приложений и при использовании в них нерегламентируемых (ad hoc) запросов. Поэтому лучше понести затраты один раз при первом выполнении курсора и принять меры к тому, чтобы в дальнейшем он при каждой возможности использовался повторно.

В этой главе, если явно не оговорено обратное, параметр CURSOR_SHARING во всех примерах установлен в значение EXACT. Сравнение точного и неточного соответствий см. ниже в разделе «Алгоритмы сопоставления».

В следующих разделах мы объясним, как Oracle принимает решение о повторном использовании курсора. Эти знания будут исключительно полезны при планировании повторного использования курсора. К сожалению, многие пишущие на PL/SQL разработчики находятся в блаженном неведении даже о существовании такой концепции, поэтому администраторам вдвойне необходимо понимание принципов повторного использования курсоров и последствий этого использования. Сначала мы рассмотрим некоторые детали алгоритма хеширования Oracle, а затем перейдем к нюансам повторного использования курсоров. Рекомендуем вам прочитать весь раздел, прежде чем встраивать в свое приложение механизмы (реальные или предполагаемые) повторного использования.

Как Oracle принимает решение о совместном использовании

Чтобы определить, может ли планируемый к выполнению курсор воспользоваться уже скомпилированной версией из разделяемого пула, Oracle применяет сложный алгоритм. Вот его упрощенное изложение:

1. Рассчитать сумму ASCII-значений всех символов курсора (исключая переменные связывания). Например, сумма ASCII-значений следующего курсора равна 2556:

   ```
   SELECT order_date FROM orders
   ```

   Это значение рассчитывается как ASCII(S) + ASCII(E) + ASCII(L)... или 83 + 69 + 76 и т. д.
2. Применить алгоритм хеширования к полученной сумме.
3. Проверить наличие в разделяемом пуле курсора с таким же значением хеша.
4. Если такой курсор найден, он может быть использован повторно.

Обратите внимание: мы говорим «может» быть использован повторно. В большинстве случаев совпадения ASCII-хеша достаточно, но не всегда. Пояснения приводятся в этом разделе далее.

В пункте 1 говорится, что используются ASCII-значения *всех* символов. Поэтому при планировании повторного использования курсора надо учитывать даже такую мелочь, как регистр символов. Рассмотрим два курсора, выполняющихся непосредственно в SQL*Plus.

```
SQL> SELECT order_date
  2    FROM orders
  3   WHERE order_number = 11;

ORDER_DAT
---------
03-MAY-05

SQL> SELECT order_date
  2    FROM orders
  3   WHERE order_numbeR = 11;

ORDER_DAT
---------
03-MAY-05
```

Очевидно, что оба они делают в точности одно и то же, – извлекают дату заказа с номером 11. Единственное отличие заключается в том, что во втором курсоре использована прописная буква R. Этого достаточно, чтобы Oracle посчитал их разными, в результате чего в разделяемом пуле окажутся оба курсора.

```
SQL> SELECT sql_text,
  2         parse_calls,
```

```
  3         executions
  4    FROM v$sql
  5   WHERE INSTR(UPPER(sql_text),'ORDERS')  > 0
  6     AND INSTR(UPPER(sql_text),'SQL_TEXT') = 0
  7     AND command_type = 3;

SQL_TEXT                       PARSE_CALLS EXECUTIONS
------------------------------ ----------- ----------
SELECT order_date   FROM order           1          1
s  WHERE order_numbeR = 11

SELECT order_date   FROM order           1          1
s  WHERE order_number = 11
```

Столбец EXECUTIONS показывает, сколько раз выполнялся определенный курсор, а столбец PARSE_CALLS – сколько раз курсор подвергался разбору. Оба курсора потребовали полного разбора, так как их суммы ASCII-значений не совпали.

Такой подход может показаться слишком строгим и безжалостным, но таким он и должен быть, потому что база данных не может позволить себе тратить время на предварительный анализ или переформатирование курсора. У нее есть дела поважнее, например выполнение вашего приложения. В последних версиях Oracle появилась возможность переформатирования литералов в курсорах, способствующая их повторному использованию (см. раздел «Алгоритмы сопоставления» ниже в той главе), но это чревато дополнительной нагрузкой при выполнении запросов с литералами.

Один из наиболее удачных способов воспользоваться преимуществами автоматического переформатирования курсоров и стимулировать их повторное использование заключается в помещении их в PL/SQL, как показано в следующем анонимном блоке:

```
DECLARE
  CURSOR one IS
  SELECT order_date
    FROM orders
   WHERE order_number = 11;
  CURSOR two IS
  SELECT order_date
    FROM orders
   WHERE order_numbeR = 11;
  v_date DATE;
BEGIN
  -- открытие и закрытие корректного курсора
  OPEN one;
  FETCH one INTO v_date;
  CLOSE one;
  -- открытие и закрытие отличающегося курсора
  OPEN two;
  FETCH two INTO v_date;
```

```
  CLOSE two;
END;
```

Компилятор PL/SQL переформатирует отличающиеся курсоры, и в разделяемом пуле они будут представлены одним экземпляром. На первый взгляд улучшение незначительно, но будучи реализовано в масштабе всего приложения, оно может дать существенный выигрыш в производительности, так как защелкам разделяемого пула придется отслеживать меньшее число курсоров.

```
SQL_TEXT                       PARSE_CALLS EXECUTIONS
------------------------------ ----------- ----------
SELECT ORDER_DATE FROM ORDERS            2          2
WHERE ORDER_NUMBER = 11
```

Данная возможность доступна начиная с Oracle8*i* Database вплоть до Oracle Database 10*g* Release 2 с досадным пробелом в Oracle9*i* Database Release 2.

Нам представляется, что из двух компиляторов – PL/SQL и SQL – первый ведет себя более снисходительно. Тем не менее каждое выполнение нашего автономного блока влечет за собой проверку и разбор каждого из курсоров. Поэтому, даже если мы обойдемся одним курсором, счетчик полных разборов будет постоянно расти. Десять выполнений нашего автономного блока вызовут 20 полных разборов:

```
SQL_TEXT                       PARSE_CALLS EXECUTIONS
------------------------------ ----------- ----------
SELECT ORDER_DATE FROM ORDERS           20         20
WHERE ORDER_NUMBER = 11
```

Хорошая новость заключается в том, что PL/SQL позволяет исключить практически все полные разборы простым перенесением курсоров в хранимые процедуры, как в следующем примере:

```
CREATE OR REPLACE PROCEDURE simple_demo AS
  CURSOR one IS
  SELECT order_date
    FROM orders
   WHERE order_number = 11;
  CURSOR two IS
  SELECT order_date
     FROM orders
   WHERE order_numbeR = 11;
  v_date DATE;
BEGIN
  -- открытие и закрытие корректного курсора
  OPEN one;
  FETCH one INTO v_date;
  CLOSE one;
  -- открытие и закрытие отличающегося курсора, позволяющее
  -- PL/SQL его переформатировать!
```

```
  OPEN two;
  FETCH two INTO v_date;
  CLOSE two;
END;
```

После десятикратного выполнения этой процедуры в SQL*Plus, например, так:

```
SQL> BEGIN
  2   simple_demo;
  3  END;
```

в SGA будут следующие результаты:

```
SQL_TEXT                       PARSE_CALLS EXECUTIONS
------------------------------ ----------- ----------
SELECT ORDER_DATE FROM ORDERS            2         20
WHERE ORDER_NUMBER = 11
```

Эти два разбора имели место при первой компиляции процедуры. После этого процедура и все ее содержимое (включая курсоры) считаются корректными и не требуют повторного разбора.

Простое перемещение двух этих курсоров в PL/SQL позволяет воспользоваться двумя его ключевыми возможностями, особенно полезными администраторам БД:

- Компилятор PL/SQL способствует повторному использованию, будучи более снисходителен к структуре курсоров. Степень этой снисходительности обсуждается в следующем разделе.
- После того как курсор был скомпилирован в составе процедуры, пакета или функции PL/SQL, он автоматически считается разобранным и действительным до тех пор, пока остаются действительными процедура, пакет или функция.

Переформатирование курсора PL/SQL

Как уже упоминалось, компилятор PL/SQL прилагает дополнительные усилия, способствующие повторному использованию курсора, отыскивая мелкие отличия типа лишних пробелов, изменений регистра и переводов строки. Например, для следующей процедуры в разделяемом пуле будет создан единственный скомпилированный курсор:

```
CREATE OR REPLACE PROCEDURE forgiveness IS
  -- define two poorly structured cursors
  CURSOR curs_x IS
  SELECT order_date FROM orders;
  CURSOR curs_y IS
  SELECT order_date
    FROM     orders;
BEGIN
  -- позволим PL/SQL использовать его дар переформатирования
  OPEN curs_x;
```

```
  CLOSE curs_x;
  OPEN curs_y;
  CLOSE curs_y;
END;
```

Такой предварительный разбор выполняется в PL/SQL для всех курсоров, что позволяет обнаружить совпадения для всего хранимого кода. Например, для приведенного ниже курсора будет использован скомпилированный курсор из процедуры `forgiveness`.

```
CURSOR curs_x IS
SELECT order_date FROM ORDers;
```

Литералы

Еще один фактор, который надо учитывать при планировании повторного использования курсора, – это использование литералов. Рассмотрим простой фрагмент кода, выполняющий два элементарных запроса. Обратите внимание на то, что тексты запросов отличаются только номерами заказов, заданными литералами.

```
CREATE OR REPLACE PROCEDURE two_queries
AS
  v_order_date DATE;
BEGIN
  -- get order 100
  SELECT order_date
    INTO v_order_date
    FROM orders
   WHERE order_number = 100;
  -- get order 200
  SELECT order_date
    INTO v_order_date
    FROM orders
   WHERE order_number = 200;
END;
```

После первого выполнения в разделяемом пуле окажутся два курсора.

```
SQL_TEXT                       PARSE_CALLS EXECUTIONS
------------------------------ ----------- ----------
SELECT ORDER_DATE FROM ORDERS            1          1
WHERE ORDER_NUMBER = 100

SELECT ORDER_DATE FROM ORDERS            1          1
WHERE ORDER_NUMBER = 200
```

Каждый из курсоров был однократно разобран и выполнен, так как их ASCII-суммы не совпали. Более того, шанс быть повторно использованными появится у них только при выполнении запроса с явно указанным номером заказа 100 или 200. Такой подход далек от оптимального: при обработке десятков тысяч заказов потребуется выполнять десятки тысяч полных разборов.

Простейший способ добиться повторного использования этих курсоров с помощью PL/SQL заключается в их параметризации, как показано в следующем фрагменте:

```
CREATE OR REPLACE PROCEDURE two_queries AS
  -- определяем параметризованный курсор
  CURSOR get_date ( cp_order NUMBER ) IS
  SELECT order_date
    FROM orders
   WHERE order_number = cp_order;
  v_order_date DATE;
BEGIN
  -- get order 100
  OPEN get_date(100);
  FETCH get_date INTO v_order_date;
  CLOSE get_date;
  -- get order 200
  OPEN get_date(200);
  FETCH get_date INTO v_order_date;
  CLOSE get_date;
END;
```

В разделяемом пуле после его очистки и выполнения новой функции будет следующее:

```
SQL_TEXT                       PARSE_CALLS EXECUTIONS
------------------------------ ----------- ----------
SELECT ORDER_DATE FROM ORDERS            1          2
WHERE ORDER_NUMBER = :B1
```

Разделяемый пул содержит единственный курсор со счетчиком выполнений, равным 2, что говорит о том, что он уже был повторно использован.

Теперь любой другой код, выполняющий такой же запрос для получения даты заказа по его номеру, может воспользоваться уже скомпилированной версией. Например, следующая процедура будет использовать скомпилированную версию данного курсора:

```
CREATE OR REPLACE PROCEDURE another_two_queries AS
  -- тот же курсор, что и в предыдущем примере,
  -- отличается только имя параметра
  CURSOR get_date ( cp_oid NUMBER ) IS
  SELECT order_date
    FROM orders
   WHERE order_number = cp_oid;
  v_order_date DATE;
BEGIN
  -- получить заказ 300
  OPEN get_date(300);
  FETCH get_date INTO v_order_date;
  CLOSE get_date;
  -- получить заказ 400
```

```
  OPEN get_date(400);
  FETCH get_date INTO v_order_date;
  CLOSE get_date;
END;
```

После выполнения этой процедуры в разделяемом пуле будут такие данные:

```
SQL_TEXT                       PARSE_CALLS EXECUTIONS
------------------------------ ----------- ----------
SELECT ORDER_DATE FROM ORDERS            2          4
WHERE ORDER_NUMBER = :B1
```

Скомпилированный ранее курсор был использован второй процедурой. Потребовался только частичный разбор. Из этого примера видно, что хорошей практикой является отказ от использования литералов в курсорах везде, где это возможно.

Но что если приложение не может быть модифицировано из-за отсутствия исходных текстов или из-за недостатка времени или денег? В таких ситуациях Oracle может кое-чем помочь, о чем и рассказывается в следующем разделе.

Алгоритмы сопоставления

По умолчанию повторное использование курсоров в базе данных требует их *полного совпадения*. За пределами PL/SQL это правило не терпит компромиссов – ASCII-значения должны совпадать в точности. Есть только черное и белое, и никаких полутонов. Курсоры или совпадают, или нет. В PL/SQL компилятор, переформатируя курсоры, старается помочь их повторному использованию, но это все, что он может сделать. Это ограничение особенно досадно, когда курсоры отличаются лишь текстом литералов. Рассмотрим два курсора:

```
SELECT order_date FROM orders WHERE order_number = '1';
SELECT order_date FROM orders WHERE order_number = '4';
```

Оба они выполняют одно и то же действие, извлекая значение `order_date` для заданного заказа, а сумма их ASCII-значений отличается всего на 3 единицы. Но стараниями алгоритма точного сопоставления оба удостаиваются собственного полного разбора и места в разделяемом пуле (даже если находятся в коде PL/SQL).

```
SQL_TEXT                       PARSE_CALLS EXECUTIONS
------------------------------ ----------- ----------
SELECT order_date FROM orders            1          1
WHERE order_number = '4'

SELECT order_date FROM orders            1          1
WHERE order_number = '1'
```

В конце концов, разделяемый пул может быть заполнен аналогичными курсорами, для которых повторное выполнение так близко – и все еще так далеко. Такое поведение в отношении литералов в курсорах

свойственно отдельным коммерческим приложениям, а также ранним версиям ODBC. С учетом такого поведения в Oracle добавлен второй алгоритм совместного использования курсоров – *сопоставление подобных* (*similar matching*). «Подобные» означает здесь, что от курсоров требуется совпадение ASCII-сумм, не учитывающих литералы.

Применяемый алгоритм совместного использования определяется параметром инициализации CURSOR_SHARING:

```
SQL> SELECT name,
  2         value
  3    FROM v$parameter
  4   WHERE name = 'cursor_sharing';

NAME                 VALUE
-------------------- ---------------
cursor_sharing       EXACT
```

Наряду с заданием данного параметра для БД в целом вы можете определить его для отдельного сеанса при помощи команды ALTER SESSION.

```
SQL> ALTER SESSION SET cursor_sharing = SIMILAR;

Session altered.
```

Вот что находится в разделяемом пуле после выполнения отличающихся литералами курсоров в случае применения алгоритма сопоставления подобных:

```
SQL_TEXT              PARSE_CALLS EXECUTIONS
--------------------- ----------- ----------
SELECT order_date FRO           2          2
M orders WHERE order_
number = :"SYS_B_0"
```

Явно заданные значения были преобразованы в переменные связывания, что значительно повысило вероятность повторного использования.

Значение SIMILAR не означает, что Oracle будет слепо подставлять переменные связывания вместо каждого найденного им литерала. Например, он не станет делать этого, если в результате значительно изменится план выполнения, выбранный оптимизатором по стоимости – как в приведенном ниже примере с существенно асимметричным распределением заказов по регионам.

```
SQL> SELECT region_id,
  2         count(*)
  3    FROM orders
  4  GROUP BY region_id;

 REGION_ID   COUNT(*)
---------- ----------
         1       9999
         2          1
```

Если оптимизатор располагает актуальной статистикой, Oracle учтет эту асимметрию и выберет разные планы выполнения для получения записей по каждому из регионов. Для региона 1 он выберет полный просмотр таблицы, так как ему надо просмотреть все записи кроме одной. Для региона 2 он предпочтет выборку одной строки с помощью индекса по полю REGION_ID. Выполним эти запросы с включенным режимом AUTOTRACE, чтобы продемонстрировать сказанное.

```
SQL> SELECT COUNT(*)
  2    FROM ( SELECT *
  3             FROM orders
  4            WHERE region_id = 1 );

  COUNT(*)
----------
      9999

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=3 Card=1 Bytes=2)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'ORDERS' (TABLE) (Cost=3 Card=21 Bytes=42)

SQL> SELECT COUNT(*)
  2    FROM ( SELECT *
  3             FROM orders
  4            WHERE region_id = 2 );

  COUNT(*)
----------
         1

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1 Card=1 Bytes=2)
   1    0   SORT (AGGREGATE)
   2    1    INDEX (RANGE SCAN) OF 'ORDER_REGION' (INDEX) (Cost=1 Card=1 Bytes=2)
```

В действительности нас интересуют находящиеся в разделяемом пуле курсоры.

```
SQL_TEXT              PARSE_CALLS EXECUTIONS
--------------------- ----------- ----------
SELECT COUNT(*)   FRO           1          1
M ( SELECT *
   FROM orders
    WHERE region_id =
 :"SYS_B_0" )

SELECT COUNT(*)   FRO           1          1
M ( SELECT *
   FROM orders
    WHERE region_id =
 :"SYS_B_0" )
```

Несмотря на то что тексты курсоров после подстановки идентичны, был создан отдельный курсор, так как Oracle обнаружил слишком большие изменения в плане выполнения. Благодаря такой стратегии производительность не страдает при установке режима SIMILAR для совместного использования курсора.

Параметр CURSOR_SHARING может принимать еще и третье значение – FORCE. В соответствии со своим наименованием оно означает принудительное повторное использование курсоров при совпадении их текстов после замены литералов на переменные связывания. При этом методе выполняется прямая подстановка, непосредственно на основании которой оптимизатор по стоимости (CBO – cost-based optimizer) строит план выполнения. Такой путь не всегда оптимален, так как, зная действительное значение, оптимизатор CBO мог бы принять лучшее решение – как было показано в примере с REGION_ID.

После выполнения запроса с подсчетом регионов 1 и 2 с использованием значения FORCE в разделяемом пуле будет находиться только один курсор.

```
SQL_TEXT              PARSE_CALLS EXECUTIONS
--------------------- ----------- ----------
SELECT COUNT(*)   FRO           2          2
M ( SELECT *
   FROM orders
    WHERE region_id =
 :"SYS_B_0" )
```

Значение FORCE появилось в Oracle8*i* Database (8.1.6) еще до того, как в Oracle9*i* Database появилось значение SIMILAR. Режим SIMILAR был введен после того, как многие администраторы и разработчики сочли метод FORCE слишком грубым в отношении производительности запросов (как было показано в примере выше). Если вы решите воспользоваться одним из этих режимов, мы настоятельно рекомендуем вам основательно протестировать производительность, чтобы убедиться в том, что преимущества совместного использования курсоров не сводятся на нет снижением производительности запросов.

Помните: несмотря на удобство использования режимов SIMILAR и FORCE, они не могут заменить хорошо продуманного и логичного использования курсоров в коде ваших программ.

Совпадения текстов может быть недостаточно

Помимо совпадения текстов курсоров есть еще ряд факторов, влияющих на повторное использование курсоров – например, несовпадения, определяемые статистикой оптимизатора или настройкой архитектуры Globalization Support (поддержка глобализации, прежде называвшаяся NLS – National Language Support, поддержка национальных языков). В таких ситуациях простого совпадения ASCII-значений недостаточно.

Рассмотрим пример установки режима работы оптимизатора.

```
SQL> ALTER SESSION SET optimizer_mode = FIRST_ROWS;

Session altered.

SQL> SELECT COUNT(*)
  2    FROM orders;

  COUNT(*)
----------
     10000

SQL> ALTER SESSION SET optimizer_mode = ALL_ROWS;

Session altered.

SQL> SELECT COUNT(*)
  2    FROM orders;

  COUNT(*)
----------
     10000
```

Квалифицированные администраторы знают, что в этом случае будут созданы два курсора: тексты совпадают, но использованы разные режимы оптимизации, поэтому Oracle строит для курсоров разные планы выполнения. Вот что находится в разделяемом пуле:

```
SQL> SELECT sql_id,
  2         sql_text,
  3         parse_calls,
  4         executions
  5    FROM v$sql
  6   WHERE INSTR(UPPER(sql_text),'ORDERS')  > 0
  7     AND INSTR(UPPER(sql_text),'SQL_TEXT') = 0
  8     AND command_type = 3;

SQL_ID        SQL_TEXT              PARSE_CALLS EXECUTIONS
------------- --------------------- ----------- ----------
d8ksp6aaxa26d SELECT COUNT(*)   FRO           1          1
              M orders

d8ksp6aaxa26d SELECT COUNT(*)   FRO           1          1
              M orders
```

Начиная с Oracle Database 10*g* Release 1 для однозначной идентификации курсора в таких представлениях, как V$SQL и V$OPEN_CURSOR, используется столбец SQL_ID. В более ранних версиях для этой цели применялась комбинация столбцов HASH_VALUE и ADDRESS.

Очевидно, Oracle обнаружил, что курсоры идентичны, так как присвоил им одинаковые идентификаторы, но сделал второй курсор потомком первого.

В данном случае мы знаем, что два курсора потребовались из-за различных режимов оптимизации, но что если бы этой информации у нас не было? Как мы можем понять, зачем понадобился второй курсор? Обратимся к представлению V$SQL_SHARED_CURSOR:

```
SQL> SELECT sql_id,
  2         child_number,
  3         optimizer_mismatch
  4    FROM v$sql_shared_cursor
  5   WHERE sql_id = 'd8ksp6aaxa26d';

SQL_ID        CHILD_NUMBER O
------------- ------------ -
d8ksp6aaxa26d            0 N
d8ksp6aaxa26d            1 Y
```

Значение «Y» во втором столбце означает: да, дочерний курсор понадобился из-за различий в оптимизаторе. Мы намеренно ограничились в этом запросе столбцом OPTIMIZER_MISMATCH, чтобы показать причину невозможности повторного использования курсора. На самом деле представление V$SQL_SHARED_CURSOR содержит множество полей (например, в Oracle Database 10*g* Release 1 их 39), каждое из которых соответствует определенной возможной причине отказа от повторного использования.

Меня всегда интересовало, почему это представление не называется V$SQL_UNSHARED_SQL_CURSOR, так как оно показывает именно это. Так или иначе, это представление очень удобно для диагностики повторного использования курсоров, поэтому рекомендую вам чаще обращаться к нему.

Сравнение явных и неявных курсоров

Проблема выбора между явными и неявными курсорами породила многолетнюю дискуссию – в общем, о выборе между конструкциями «OPEN, FETCH, CLOSE» и «SELECT INTO». В этом разделе мы не будем касаться вопросов производительности, так как в последних версиях Oracle проделана большая работа для снятия остроты этой проблемы Вместо этого сосредоточимся на том, что происходит в базе данных, и обсудим различия в применении этих курсоров в PL/SQL, включая и тот факт, что они не всегда совпадают в разделяемом пуле.

В чем отличие?

В PL/SQL *неявные курсоры* – это курсоры, которые определяются в момент выполнения. Вот пример:

```
DECLARE
  v_date DATE;
BEGIN
  SELECT order_date
```

```
    INTO v_date
    FROM orders
   WHERE order_number = 100;
END;
```

В ходе выполнения этого кода создается курсор для выборки значения `order_date` для заказа с номером 100. Таким образом, курсор был неявно определен во время выполнения кода.

Явный курсор – это курсор, который определяется до начала выполнения. Вот простой пример:

```
DECLARE
  CURSOR curs_get_od IS
  SELECT order_date
    FROM orders
   WHERE order_number = 100;
  v_date DATE;
BEGIN
  OPEN curs_get_od;
  FETCH curs_get_od INTO v_date;
  CLOSE curs_get_od;
END;
```

Неявный курсор выглядит гораздо проще и короче в записи, поэтому первым побуждением может быть выбор именно этого варианта. Однако явные курсоры имеют ряд преимуществ, оправдывающих удлинение кода PL/SQL; их рассмотрению посвящены следующие два раздела.

Атрибуты курсора

Ключевое преимущество явного курсора заключается в наличии у него атрибутов, облегчающих применение условных операторов. Рассмотрим следующий пример: мы хотим найти заказ и, если он найден, выполнить некоторые действия. Первая процедура, использующая неявный курсор, вынуждена заниматься перехватом исключений, чтобы определить, была ли найдена запись.

```
CREATE OR REPLACE PROCEDURE demo AS
  v_date      DATE;
  v_its_there BOOLEAN := TRUE;
BEGIN
  BEGIN
    SELECT order_date
      INTO v_date
      FROM orders
     WHERE order_number = 1;
  EXCEPTION
    WHEN no_data_found THEN
      v_its_there := FALSE;
    WHEN OTHERS THEN
      RAISE;
```

```
  END;
  IF NOT v_its_there THEN
    do_something;
  END IF;
END;
```

Вторая процедура, написанная с использованием явного курсора, выглядит гораздо понятнее, так как наличие у курсора атрибута %NOTFOUND делает очевидным проверяемое условие. Отсутствует также необходимость в дополнительном блоке BEGIN–END, нужном только для реализации логики.

```
CREATE OR REPLACE PROCEDURE demo AS
  CURSOR curs_get_date IS
  SELECT order_date
    FROM orders
   WHERE order_number = 1;
  v_date DATE;
BEGIN
  OPEN curs_get_date;
  FETCH curs_get_date INTO v_date;
  IF curs_get_date%NOTFOUND THEN
    do_something;
  END IF;
  CLOSE curs_get_date;
END;
```

Oracle поддерживает следующие атрибуты курсоров:

Атрибут	Описание
%BULK_ROWCOUNT	Количество записей, возвращаемых операцией массовой выборки (BULK COLLECT INTO).
%FOUND	TRUE, если последняя операция FETCH была успешной; FALSE — в противном случае.
%NOTFOUND	TRUE, если последняя операция FETCH была неуспешной; FALSE — в противном случае.
%ISOPEN	TRUE, если курсор открыт; FALSE — в противном случае.
%ROWCOUNT	Количество записей, выбранных на текущий момент курсором.

Вы, возможно, знаете, что некоторые из этих атрибутов доступны также для неявных курсоров. Однако программную логику гораздо удобнее реализовывать с помощью явных курсоров, особенно когда в вашей программе несколько курсоров, как показано в этом коротком примере:

```
IF curs_get_order%ROWCOUNT = 1 THEN
  IF curs_get_details%FOUND THEN
    process_order_detail;
```

Параметры курсора

Как уже говорилось в этой главе, параметризация курсоров помогает повысить степень их повторного использования. Вот простой пример процедуры получения даты заказа с параметризованным курсором:

```
DECLARE
  CURSOR curs_get_od ( cp_on NUMBER ) IS
   SELECT order_date
     FROM orders
    WHERE order_number = cp_on;
   v_date DATE;
 BEGIN
   OPEN curs_get_od(100);
  FETCH curs_get_od INTO v_date;
  CLOSE curs_get_od;
END;
```

Если далее в этой программе нам понадобятся данные о заказах с номерами 200, 300 и 500, достаточно будет еще раз открыть этот же курсор. Такой подход способствует повторному использованию курсоров как в самой PL/SQL-программе, так и в разделяемом пуле.

Несовпадение в разделяемом пуле

Явные и неявные курсоры в разделяемом пуле не совпадают. Поясним на примере, что имеется в виду.

```
DECLARE
  CURSOR get_region IS
  SELECT region_id FROM orders WHERE region_id = 2;
  v_region NUMBER;
BEGIN
  OPEN get_region;
  FETCH get_region INTO v_region;
  CLOSE get_region;
  SELECT region_id INTO v_region FROM orders WHERE region_id = 2;
END;
```

Сколько курсоров будет в разделяемом пуле? Ответ: два.

```
SQL_TEXT                       PARSE_CALLS EXECUTIONS
------------------------------ ----------- ----------
SELECT REGION_ID FROM ORDERS W           1          1
HERE REGION_ID = 2

SELECT REGION_ID FROM ORDERS W           1          1
HERE REGION_ID = 2
```

Несмотря на то что эти курсоры выглядят одинаково и оба находятся в разделяемом пуле, с точки зрения Oracle у них достаточно отличий (может быть, предложение INTO?), чтобы хранить их отдельно. Поэтому не следует надеяться, что явный и неявный курсоры совпадут в разделяемом пуле. Лучше придерживаться какого-то одного способа.

Мягкое закрытие курсора

При разработке приложений я всегда стараюсь использовать любые возможности, приводящие к повышению производительности. Я всегда задаю себе вопросы – например, если функция многократно вызывается в течение одного сеанса Oracle, почему бы мне не кэшировать какие-то общие данные вместо того, чтобы каждый раз запрашивать их? Или, если мне известно, что базовые данные меняются один раз в час, почему бы не сохранить результаты запроса и тем самым избежать повторных запросов в течение последующих 60 минут?

Одно из предположений Oracle заключается в том, что если курсор был однажды использован в сеансе, то он когда-нибудь будет использован повторно (даже если был явно закрыт). Прием, реализующий это предположение, называется *мягким закрытием*, или, как я это называю, «закрытием без закрытия».

Рассмотрим простой пример с неявным курсором.

```
SQL> SELECT NULL
  2    FROM DUAL;

N
-

1 row selected
```

Неявный курсор был создан, открыт, использован для выборки и закрыт – все в одном операторе SELECT. Теперь он должен быть отсоединен от сеанса, в котором выполнялся, правильно? Не торопитесь. Для того чтобы воспользоваться преимуществами возможного повторного использования, выполняется лишь мягкое закрытие курсора, которое позволяет ускорить его повторное выполнение в этом сеансе.

Курсоры, связанные с определенным сеансом, перечислены в представлении V$OPEN_CURSOR. Здесь содержатся как открытые в данный момент курсоры, так и те, которые были мягко закрыты. Вот что находится в этом представлении для сеанса, выполняющего запрос к DUAL:

```
SQL> SELECT sql_text
  2    FROM v$open_cursor
  3   WHERE sid = 43;

SQL_TEXT
------------------------
SELECT NULL   FROM DUAL
```

Такое мягкое закрытие происходит при явном закрытии курсора оператором CLOSE или при неявном закрытии в момент выхода из области видимости.

Для того чтобы все курсоры не остались навечно в состоянии мягкого закрытия, можно установить в параметре инициализации базы данных OPEN_CURSORS предельное количество курсоров для сеанса. При за-

полнении списка наиболее долго не использовавшийся закрытый курсор выбрасывается, а на его место записывается новый. Однако если сеанс попытается явно открыть больше курсоров, чем указано в этом параметре, он получит ошибку ORA-01000: maximum open cursors exceeded (превышено максимальное число открытых курсоров).

Открытие явных и неявных курсоров

Еще одно место, где явные и неявные курсоры трактуются по-разному, – это список открытых курсоров. Рассмотрим это на примере, где значение параметра OPEN_CURSORS равно 20. Сначала выполним несколько неявных курсоров:

```
DECLARE
  v_dummy varchar2(10);
BEGIN
  SELECT 'A' INTO v_dummy  FROM orders;
  SELECT 'B' INTO v_dummy  FROM orders;
...и так далее для всех прописных и строчных букв алфавита...
  SELECT 'x' INTO v_dummy  FROM orders;
  SELECT 'y' INTO v_dummy  FROM orders;
  SELECT 'z' INTO v_dummy  FROM orders;
END;
```

Список связанных с сеансом курсоров будет выглядеть так:

```
SQL> SELECT oc.sql_text
  2    FROM v$open_cursor oc,
  3         v$sql         sq
  4   WHERE user_name = 'DRH'
  5     AND oc.sql_id = sq.sql_id
  6     AND command_type = 3;

SQL_TEXT
--------------------------------
SELECT 'n' FROM ORDERS
SELECT 'z' FROM ORDERS
SELECT 'o' FROM ORDERS
SELECT 'q' FROM ORDERS
SELECT 'x' FROM ORDERS
SELECT 'l' FROM ORDERS
SELECT 'v' FROM ORDERS
SELECT 's' FROM ORDERS
SELECT 'p' FROM ORDERS
SELECT 'w' FROM ORDERS
SELECT 'm' FROM ORDERS
SELECT 'u' FROM ORDERS
SELECT 'k' FROM ORDERS
SELECT 'j' FROM ORDERS
SELECT 'i' FROM ORDERS
SELECT 'y' FROM ORDERS
SELECT 'r' FROM ORDERS
SELECT 't' FROM ORDERS
```

```
SELECT 'h' FROM ORDERS

19 rows selected.
```

Только последние курсоры остались в состоянии мягкого закрытия. Остальные были удалены, чтобы освободить место для новых курсоров.

Теперь выполним несколько явных курсоров, открывая и закрывая каждый из них.

```
DECLARE
  CURSOR curs_65 IS SELECT 'A' FROM orders;
  CURSOR curs_66 IS SELECT 'B' FROM orders;
...и так далее для всех прописных и строчных букв алфавита...
...ASCII-код от 65 до 122
  CURSOR curs_122 IS SELECT 'z' FROM orders;
BEGIN
  OPEN curs_65;
  CLOSE curs_65;
...и так далее...
  OPEN curs_122;
  CLOSE curs_122;
END;
```

В результате получим список связанных с сеансом курсоров, очень похожий на тот, который мы видели в случае с неявными курсорами. Но представьте, что будет, если мы поленимся и не закроем все явные курсоры, как в следующем примере.

```
DECLARE
  CURSOR curs_65 IS SELECT 'A' FROM orders;
  CURSOR curs_66 IS SELECT 'B' FROM orders;
...и так далее для всех прописных и строчных букв алфавита...
...ASCII-код от 65 до 122
  CURSOR curs_122 IS SELECT 'z' FROM orders;
BEGIN
  OPEN curs_65;
  OPEN curs_66;
...и так далее...
  OPEN curs_122;
END;
```

Где-то на 20-м курсоре мы получим ошибку, так как сеанс попытается выйти за отведенное для него ограничение в 20 открытых курсоров.

```
ERROR at line 1:
ORA-01000: maximum open cursors exceeded
ORA-06512: at line 21
ORA-06512: at line 80
```

Это одна из причин, по которым так важно всегда закрывать явные курсоры.

Так каким же должно быть разумное значение параметра OPEN_CURSORS? Ответ прост: столько, сколько нужно, плюс один. На первый взгляд от

такого ответа мало пользы, но для установки этого параметра нет других практических рекомендаций. Если значение слишком мало, программа не выполнится из-за ошибки ORA-1000. Если значение слишком велико, курсоры (явные и неявные) могут навсегда остаться в состоянии мягкого закрытия. Хорошая новость в том, что при задании параметра пространство не резервируется, поэтому установка большого значения не приводит к перерасходу памяти.

Приведенный далее запрос возвращает подходящее значение, подсчитывая полное количество курсоров (открытых и мягко закрытых) для текущего сеанса. Я регулярно запускаю его на ранних этапах разработки приложения.

```
SYS> SELECT *
  2    FROM ( SELECT sid,
  3                  COUNT(*)
  4             FROM v$open_cursor
  5           GROUP BY sid
  6           ORDER BY COUNT(*) DESC)
  7  WHERE ROWNUM = 1;

       SID   COUNT(*)
---------- ----------
        46         20
```

Затем я устанавливаю параметр OPEN_CURSORS равным максимальному полученному значению с запасом в 10 или 20.

Динамический SQL

В динамическом SQL (NDS – Native Dynamic SQL), как правило, тоже могут быть реализованы преимущества мягкого закрытия и повторного использования курсоров, но наилучшие результаты достигаются при использовании переменных связывания. Рассмотрим две процедуры, которые выполняют одинаковые действия, но в одной использованы переменные связывания, а в другой – конкатенация.

```
CREATE OR REPLACE PROCEDURE bind ( p_on NUMBER ) AS
  v_od DATE;
BEGIN
  EXECUTE IMMEDIATE 'SELECT order_date ' ||
                    ' FROM orders '     ||
                   ' WHERE order_number = :v_on'
          INTO v_od
          USING p_on;
END;

CREATE OR REPLACE PROCEDURE concatenate ( p_on NUMBER ) AS
  v_od DATE;
BEGIN
  EXECUTE IMMEDIATE 'SELECT order_date ' ||
                    ' FROM orders '     ||
```

```
                    ' WHERE order_number = ' || p_on
          INTO v_od;
END;
```

Сначала выполним трижды версию с переменными связывания:

```
SQL> BEGIN
  2    FOR counter IN 1..3 LOOP
  3      bind(counter);
  4    END LOOP;
  5  END;
  6  /

PL/SQL procedure successfully completed.
```

В списке открытых видим уже знакомый нам курсор:

```
SELECT order_date  FROM orders
  WHERE order_number = :v_on
```

Количество разборов и выполнений, как и ожидалось, равно 1 и 3:

```
SQL_TEXT                       PARSE_CALLS EXECUTIONS
------------------------------ ----------- ----------
SELECT order_date  FROM orders           1          3
  WHERE order_number = :v_on
```

Теперь трижды выполним версию с конкатенацией:

```
SQL> BEGIN
  2    FOR counter IN 1..3 LOOP
  3      concatenate(counter);
  4    END LOOP;
  5  END;
  6  /

PL/SQL procedure successfully completed.
```

Список открытых курсоров имеет такой вид в силу того, что сохраняется только самый последний из них – в надежде на повторное использование.

```
SQL_TEXT
------------------------------
SELECT order_date  FROM orders
  WHERE order_number = 3
```

Интересно взглянуть на счетчики разборов и выполнений. Каждый из трех курсоров был один раз разобран и один раз выполнен. Очевидно, что это слишком расточительно.

```
SQL_TEXT                       PARSE_CALLS EXECUTIONS
------------------------------ ----------- ----------
SELECT order_date  FROM orders           1          1
  WHERE order_number = 3
SELECT order_date  FROM orders           1          1
  WHERE order_number = 1
```

```
SELECT order_date  FROM orders          1          1
  WHERE order_number = 2
```

Динамический SQL – это мощный и удобный инструмент. При минимально продуманном использовании он позволяет получить все преимущества повторного использования курсоров, доступные в обычном PL/SQL.

Использование курсоров не только для запросов

Вместе с развитием СУБД Oracle развиваются и курсоры. Наряду с достижениями в повышении производительности, описанными в предыдущих разделах, расширилась и функциональность курсоров, которая больше не ограничивается выполнением запросов, – курсоры теперь интегрируются в приложения на уровне проектирования и компоновки. В этом разделе описаны не упоминавшиеся ранее дополнительные возможности работы с курсорами.

Массовая выборка, тип данных `REF CURSOR`, параметры курсора и курсорные выражения – все эти инструменты весьма полезны администратору базы данных, занимающемуся исследованием и улучшением производительности приложений. Особенно полезны рассматриваемые здесь средства в тех случаях, когда вы работаете с тяжело нагруженной базой данных и для вас очень важно свести к абсолютному минимуму количество затрагиваемых вами записей. Например, ссылочный тип `REF CURSOR` можно использовать для контроля доступа к данным из клиентского приложения, которое может даже не подозревать о способе структурирования таблиц. Параметры курсора позволяют расширить доступ к данным. (В главе 3 обсуждаются и другие способы достижения этой цели.) А курсорные выражения (вложенные курсоры) активно способствуют тому, чтобы выполнялись только те операции, которые действительно должны выполняться.

Массовая выборка

Если вы будете выбирать записи по одной в цикле PL/SQL, вы столкнетесь с повышенными накладными расходами на переключение контекста между SQL и PL/SQL для каждой записи. Это существенно увеличит общее время выполнения, особенно при большом числе записей. Уменьшить количество переключений контекста можно с помощью массовой выборки (`BULK COLLECT INTO`), запрашивающей записи порциями или все сразу.

Сначала рассмотрим пример выборки записей по одной:

```
CREATE OR REPLACE PROCEDURE one_at_a_time AS
  CURSOR curs_get_ord IS
  SELECT order_number,
         order_date
    FROM orders
```

```
  ORDER BY order_number;
  v_order_number NUMBER;
  v_order_date   DATE;
BEGIN
  FOR v_order_rec IN curs_get_ord LOOP
    do_something;
  END LOOP;
END;
```

Если таблица ORDERS содержит 100 записей, то будет выполнено 100 переключений контекста. Вот вариант этой программы с массовой выборкой.

```
CREATE OR REPLACE PROCEDURE all_at_once AS
  CURSOR curs_get_ord IS
  SELECT order_number,
         order_date
    FROM orders
  ORDER BY order_number;

  -- локальная коллекция для хранения результата массовой выборки
  TYPE v_number_t IS TABLE OF NUMBER;
  TYPE v_date_t   IS TABLE OF DATE;
  v_order_number v_number_t;
  v_order_date   v_date_t;

BEGIN
  -- получить сразу все заказы
  OPEN curs_get_ord;
  FETCH curs_get_ord BULK COLLECT INTO v_order_number, v_order_date;
  CLOSE curs_get_ord;
  -- если найдены хоть какие-то заказы, обработать их в цикле
  -- по локальной коллекции
  IF NVL(v_order_number.COUNT,0) > 0 THEN
    FOR counter IN v_order_number.FIRST..v_order_number.LAST LOOP
      do_something;
    END LOOP;
  END IF;
END;
```

Для больших наборов записей выигрыш в производительности может быть огромен, поэтому мы настоятельно рекомендуем вам при любой возможности использовать этот способ.

У массовой выборки есть и другое, менее очевидное преимущество: база данных не должна заботиться о согласованности данных по чтению на всем протяжении извлечения и обработки данных. Обратимся еще раз к предыдущему примеру. Если гипотетическая процедура DO_SOMETHING тратит пять секунд на обработку каждой из 100 записей, полученных из таблицы ORDERS, серверу Oracle придется более восьми минут поддерживать согласованную по чтению копию этих данных. Если таблица ORDERS участвует во множестве других операций DML, то сегмент отката базы данных должен будет заниматься поддержанием со-

гласованного представления данных на всем протяжении этой длинной операции.

В данном случае есть одна потенциальная трудность, возникающая при переходе к массовой выборке: процедура DO_SOMETHING должна будет обрабатывать ситуации, когда записи, с которыми она собирается работать, уже не существуют, так как они были удалены за время, прошедшее с момента массовой выборки.

Альтернативный способ заключается в помещении всех полученных массовой выборкой записей в память и их последующей обработке. При таком подходе значительно уменьшаются шансы получить неприятную ошибку ORA-01555 – Snapshot Too Old (сегмент отката слишком мал).

Так как массовая выборка помещает записи в память сеанса, следует учитывать ограничения на доступный сеансу объем памяти. Если для приложения критичны затраты памяти сеанса, то вы можете использовать предложение LIMIT для ограничения количества одновременно запрашиваемых записей. Например, так:

```
OPEN curs_get_ord;
LOOP
  -- получить следующие 1000 заказов
  FETCH curs_get_ord BULK COLLECT INTO v_order_number, v_order_date LIMIT 1000;
  -- если какие-либо заказы найдены, то пройти по ним в цикле
  IF NVL(v_order_number.COUNT,0) > 0 THEN
    FOR counter IN v_order_number.FIRST..v_order_number.LAST LOOP
      do_something;
    END LOOP;
  ELSE
    EXIT;
  END IF;
END LOOP;
CLOSE curs_get_ord;
```

Я часто пользуюсь массовой выборкой при запросах к таблицам производительности Oracle (V$), так как меньше всего я хотел бы, чтобы БД занималась дополнительной работой только ради того, чтобы я мог посмотреть, например, сколько операций чтения и записи выполнил каждый из сеансов. Вот алгоритм, которого я придерживаюсь:

```
BEGIN
  массовая выборка текущих сеансов из V$SESSION
  для каждого сеанса
    запросить статистику сеанса по чтению и записи
  end if
END;
```

Рекомендую использовать эту возможность как можно чаще при запросах к сильно нагруженным представлениям производительности Oracle.

Тип данных REF CURSOR

Переменные типа REF CURSOR могут ссылаться на любые *реальные* курсоры. Программа, использующая тип REF CURSOR, может работать с курсорами Oracle, не заботясь о том, какие конкретно данные будут извлечены ими во время выполнения. Вот очень простой пример:

```
CREATE OR REPLACE PROCEDURE ref_curs AS
  v_curs SYS_REFCURSOR;
BEGIN
  OPEN v_curs FOR 'SELECT order_number ' ||
                  ' FROM orders';
  CLOSE v_curs;
END;
```

Во время компиляции Oracle не знает, каким будет текст запроса, – он видит строковую переменную. Но наличие типа REF CURSOR говорит ему о том, что надо будет обеспечить некую работу с курсором.

Ссылки на курсоры очень полезны при создании «черных ящиков», предоставляющих другим приложениям возможности доступа к данным с помощью функций, создающих курсор и возвращающих тип REF CURSOR, как показано в этом примере:

```
CREATE OR REPLACE FUNCTION all_orders ( p_id NUMBER )
                  RETURN SYS_REFCURSOR  IS
  v_curs SYS_REFCURSOR;
BEGIN
  OPEN v_curs FOR 'SELECT * ' ||
                  ' FROM orders ' ||
                  ' WHERE order_number = ' || p_id;
  RETURN v_curs;
END;
```

Вызывающая программа передает этой функции номер заказа order_number и получает от нее доступ к соответствующим данным, ничего не зная о них заранее. Внешние приложения, например, реализованные на платформе .NET от Microsoft, могут по полученной ссылке REF CURSOR определить такие атрибуты, как имена и типы данных столбцов, чтобы выбрать подходящий для них способ отображения.

Вот как функция all_orders может быть вызвана из PL/SQL:

```
DECLARE
  v_curs      SYS_REFCURSOR;
  v_order_rec ORDERS%ROWTYPE;
BEGIN
  v_curs := all_orders(1);
  FETCH v_curs INTO v_order_rec;
  IF v_curs%FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Found It');
  END IF;
  CLOSE v_curs;
END;
```

Строгий и слабый контроль типов REF CURSOR

Есть два вида контроля типа данных REF CURSOR, строгий и слабый. Разница в том, что для слабо типизированной переменной REF CURSOR заранее неизвестно, на какой набор данных она будет ссылаться; в то время как при строгой типизации явно указывается, каким будет возвращаемый набор данных.

Тип данных SYS_REFCURSOR, использованный в двух предыдущих примерах, появился в Oracle9*i* Database. Он позволяет быстро определять переменные REF CURSOR со слабым контролем типа. В предыдущих версиях они определялись таким способом:

```
DECLARE
  TYPE v_curs_t IS REF_CURSOR;
  v_curs v_curs_t;
```

Слабо типизированные переменные REF CURSOR могут использоваться практически любыми запросами, так как они не привязаны к конкретным структурам данных.

```
DECLARE
  v_curs SYS_REFCURSOR;
BEGIN
  OPEN v_curs FOR 'SELECT order_number ' ||
                  ' FROM orders';
  CLOSE v_curs;
  OPEN v_curs FOR 'SELECT * ' ||
                  ' FROM orders';
  CLOSE v_curs;
END;
```

Фактический запрос, сопоставленный ссылке REF CURSOR, подвергается проверке и разбору и помещается в область SGA точно так же, как любой другой курсор.

```
SQL_TEXT                       PARSE_CALLS EXECUTIONS
------------------------------ ----------- ----------
SELECT *  FROM orders                    1          1
SELECT order_number  FROM orde           1          1
rs
```

Заметьте, однако, что для REF CURSOR не действует мягкое закрытие, так что они не могут воспользоваться преимуществом последующего «сверхбыстрого» открытия. То есть курсоры типа REF CURSOR будут работать медленнее, чем обычные курсоры.

Кроме того, слабо типизированные переменные REF CURSOR создают дополнительную нагрузку, когда Oracle «на лету» определяет структуру возвращаемого набора данных. Поэтому для лучшей производительности следует по возможности использовать строгий контроль типов для REF CURSOR. Вот несколько примеров строгой типизации переменных REF CURSOR:

```
DECLARE
  -- тип для записей о заказах
  TYPE v_order_curs IS REF CURSOR RETURN orders%ROWTYPE;
  v_oc v_order_curs;
  -- тип только для номеров заказов
  TYPE v_order_number_t IS RECORD ( order_number orders.order_number%TYPE );
  TYPE v_order_number_curs IS REF CURSOR RETURN v_order_number_t;
  v_ocn v_order_number_curs;
```

Попытка использовать переменную REF CURSOR для получения набора данных несоответствующего типа приведет к появлению сообщения об ошибке ORA-06550.

```
OPEN v_ocn FOR SELECT * FROM ORDERS;
               *
ERROR at line 10:
ORA-06550: line 10, column 18:
PLS-00382: expression is of wrong type
```

Атрибуты REF CURSOR

Курсоры типа REF CURSOR имеют тот же полный набор атрибутов, что и явные курсоры, как показано в примере:

```
DECLARE
  v_curs SYS_REFCURSOR;
  v_on   NUMBER;
BEGIN
  OPEN v_curs FOR 'SELECT order_number ' ||
                  ' FROM orders';
  FETCH v_curs INTO v_on;
  LOOP
    EXIT when v_curs%NOTFOUND;
    IF v_curs%ROWCOUNT = 1 THEN
      NULL;
    END IF;
    FETCH v_curs INTO v_on;
  END LOOP;
  CLOSE v_curs;
END;
```

Динамический доступ к данным

Курсоры типа REF CURSOR весьма полезны в ситуациях, когда текст запроса заранее не известен, но известна логика работы. Например, приведенная ниже процедура получит текст запроса и откроет для него курсор. После этого она передаст переменную REF CURSOR другой процедуре, которая выполнит выборку данных из курсора (и затем его закроет).

```
CREATE OR REPLACE PROCEDURE order_cancel ( p_sql  VARCHAR2 ) IS
  v_curs SYS_REFCURSOR;
BEGIN
  IF v_curs%ISOPEN THEN
```

```
    CLOSE v_curs;
  END IF;
  BEGIN
    OPEN v_curs FOR p_sql;
  EXCEPTION
    WHEN OTHERS THEN
      RAISE_APPLICATION_ERROR(-20000,'Unable to open cursor');
  END;
  order_cancel_details(v_curs);
  CLOSE v_curs;
END;
```

Тогда функция `order_cancel` может быть вызвана так:

```
BEGIN
  order_cancel('SELECT order_number FROM orders
                  WHERE due_date <= TRUNC(SYSDATE)');
END;
```

Курсор в качестве параметра

Примеры из предыдущего раздела показывают, что курсоры в качестве параметров можно передавать в виде текста SQL. Это можно также сделать, используя предложение `SELECT` с ключевым словом `CURSOR`.

```
SELECT count_valid(CURSOR(SELECT order_number
                            FROM orders
                           WHERE processed IS NULL))
  FROM dual;
```

Функция `count_valid` может выглядеть так:

```
CREATE OR REPLACE FUNCTION count_valid( p_curs SYS_REFCURSOR )
                  RETURN NUMBER IS
  v_on NUMBER;
  v_ret_val NUMBER := 0;
BEGIN
  FETCH p_curs INTO v_on;
  LOOP
    EXIT WHEN p_curs%NOTFOUND;
    IF extensive_validation(v_on) THEN
      v_ret_val := v_ret_val + 1;
    END IF;
    FETCH p_curs INTO v_on;
  END LOOP;
  RETURN(v_ret_val);
END;
```

Оператор `SELECT` передается сразу в функцию, которая перебирает в цикле возвращаемые им записи, проверяет их и возвращает количество записей, признанных верными. Это приводит к появлению двух курсоров в разделяемом пуле и в списке мягких закрытий для пользователя.

```
SQL_TEXT
----------------------------------------
SELECT "A2"."ORDER_NUMBER" "ORDER_NUMBER
" FROM "ORDERS" "A2" WHERE "A2"."PROCESSED" IS NULL

SELECT count_valid(CURSOR(SELECT order_number
                            FROM orders
                           WHERE processed IS NULL))
  FROM dual
```

Курсорные выражения

Курсорные выражения – это, по сути, вложенные курсоры. Когда мы говорим о «курсорных выражениях», мы не имеем в виду вложенные подзапросы, определяющие результирующее множество. Речь идет о вложенных запросах, возвращающих вложенное результирующее множество. Объясним это на примере.

```
SELECT order_number,
       CURSOR ( SELECT order_line_amt
                  FROM order_lines ol
                 WHERE ol.order_number = orders.order_number )
  FROM orders;
```

Этот запрос возвращает список заказов вместе с курсором, позволяющим позже запросить данные конкретного заказа. Вот как это может быть использовано в PL/SQL-процедуре:

```
/* File on web: nested_cursor.sql */
CREATE OR REPLACE PROCEDURE nested AS

  -- курсор для получения заказов с вложенным курсором
  -- для подсчета позиций в нем
  CURSOR curs_orders IS
  SELECT order_number,
         CURSOR ( SELECT order_line_amt
                    FROM order_lines ol
                   WHERE ol.order_number = orders.order_number )
    FROM orders;
  lines_curs SYS_REFCURSOR;  -- для позиций заказа
  v_order_id NUMBER;

  -- локальные переменные для массовой выборки позиций
  TYPE v_number_t IS TABLE OF NUMBER;
  v_line_amt  v_number_t;

BEGIN

  OPEN curs_orders;
  FETCH curs_orders INTO v_order_id, lines_curs;

  -- для каждого заказа...
  LOOP
    EXIT WHEN curs_orders%NOTFOUND;
```

```
    -- обрабатывать только заказы с четными номерами
    IF MOD(v_order_id,2) = 0 THEN

      -- получить сразу все позиции заказа
      FETCH lines_curs BULK COLLECT INTO v_line_amt;

      -- пройти по позициям заказа
      IF NVL(v_line_amt.COUNT,0) > 0 THEN
        FOR counter IN v_line_amt.FIRST..v_line_amt.LAST LOOP
          process_lines;
        END LOOP;
      END IF;

    END IF;  -- только заказы с четными номерами

    FETCH curs_orders INTO v_order_id, lines_curs;
  END LOOP;  -- каждый заказ

  CLOSE curs_orders;

END;
```

Курсорные выражения имеют не вполне очевидный синтаксис, но их использование дает некоторые преимущества. Главное из них заключается в том, что курсорные выражения напрямую связывают логический и физический уровни обработки как для оптимизатора Oracle, так и для программного кода. Оптимизатор извлекает выгоду из наличия явно указанной связи между двумя таблицами (ORDERS и ORDER_LINES), что позволяет ему сделать лучший выбор в тот момент, когда надо будет извлекать содержимое заказа. Программный код сам по себе ограничивает физическую работу, решая на основе логических условий, стоит ли вообще извлекать данные по определенному заказу. Поэтому не приходится запрашивать записи лишь затем, чтобы проигнорировать их в дальнейшем.

Также интересно проанализировать, что попадает в SGA после выполнения такой вложенной процедуры для 1000 заказов.

```
SQL_TEXT                       PARSE_CALLS EXECUTIONS
------------------------------ ----------- ----------
SELECT ORDER_NUMBER, CURSOR (            1          1
SELECT ORDER_LINE_AMT FROM ORD
ER_LINES WHERE ORDER_NUMBER =
ORDERS.ORDER_NUMBER ) FROM ORD
ERS

SELECT "A2"."ORDER_LINE_AMT" "         500        500
ORDER_LINE_AMT" FROM "ORDER_LI
NES" "A2" WHERE "A2"."ORDER_NU
MBER"=:CV1$
```

Обратите внимание на то, что правая часть предложения WHERE вложенного запроса превратилась в переменную связывания курсора. Так осуществляется связь с главным курсором. Заметьте также, что счетчики

разборов и выполнений для второго курсора имеют значения, равные 500, поскольку он выполнился ровно 500 раз, как и было необходимо. Еще более важно то, что доступ к данным выполнялся только 500 раз.

После того как процедура выполнена, в этом сеансе остается открытым только главный курсор. Однако во время ее выполнения открывалось множество других курсоров. Вы можете увидеть это, добавив в код 10-секундную задержку и обратившись к представлению V$OPEN_CURSORS.

```
SQL_TEXT
-----------------------------
SELECT ORDER_NUMBER, CURSOR (
SELECT ORDER_LINE_AMT FROM ORD
ER_LINES WHERE ORDER_NUMBER =
ORDERS.ORDER_NUMBER ) FROM ORD
ERS

SELECT "A2"."ORDER_LINE_AMT" "
ORDER_LINE_AMT" FROM "ORDER_LI
NES" "A2" WHERE "A2"."ORDER_NU
MBER"=:CV1$
```

Выясняется, что значение 500 для второго курсора относится к числу операций открытия до того момента, как процедура завершится и все закроет (так как курсоры выходят из области видимости). Все 500 вложенных курсоров будут использовать уже находящуюся в SGA скомпилированную версию, что видно по постоянно растущему значению счетчиков разборов и выполнений после шести выполнений вложенной процедуры.

```
SQL_TEXT                       PARSE_CALLS EXECUTIONS
------------------------------ ----------- ----------
SELECT "A2"."ORDER_LINE_AMT" "        3000       3000
ORDER_LINE_AMT" FROM "ORDER_LI
NES" "A2" WHERE :CV1$=:CV1$
```

Однако из этого описания следует, что все 500 раз преимущества мягкого закрытия остались нереализованными. Также получается, что мы очень близко подходим к разрешенному максимуму для OPEN_CURSORS. Поэтому в таких случаях лучше явно закрывать вложенный курсор, когда работа с ним закончена. (Это может быть неочевидным, так как вложенный курсор не использует явной операции открытия.) Вот измененный участок кода:

```
    -- обрабатывать только заказы с четными номерами
IF MOD(v_order_id,2) = 0 THEN

  -- неявное открытие
  FETCH lines_curs BULK COLLECT INTO v_line_amt;

    IF NVL(v_line_amt.COUNT,0) > 0 THEN
      FOR counter IN v_line_amt.FIRST..v_line_amt.LAST LOOP
```

```
        Process_lines;
    END LOOP;

END IF;  -- только заказы с четными номерами

-- закрыть вложенный курсор
CLOSE lines_curs;

END IF;
```

Уверен, к этому моменту вы уже недоумеваете, почему бы не улучшить наш пример и не переписать его с использованием единственного курсора, примерно так:

```
SELECT o.order_number,
       order_line_amt
  FROM orders      o,
       order_lines ol
 WHERE ol.order_number = o.order_number;
```

Затем должен следовать PL/SQL-код проверки делимости номера заказа надвое без остатка. Различие в этих двух подходах заключается в количестве строк, обработанных при выполнении запроса. В варианте с вложенным курсором получаем следующие значения:

```
SQL_TEXT                       ROWS_PROCESSED
------------------------------ --------------
SELECT ORDER_NUMBER, CURSOR (           1000
SELECT ORDER_LINE_AMT FROM ORD
ER_LINES OL WHERE OL.ORDER_NUM
BER = ORDERS.ORDER_NUMBER ) FR
OM ORDERS

SELECT "A2"."ORDER_LINE_AMT" "          5000
ORDER_LINE_AMT" FROM "ORDER_LI
NES" "A2" WHERE "A2"."ORDER_NU
MBER"=:CV1$
```

Другой подход, с одним курсором, даст такое значение:

```
SQL_TEXT                       ROWS_PROCESSED
------------------------------ --------------
SELECT O.ORDER_NUMBER, ORDER_L         10000
INE_AMT FROM ORDERS O, ORDER_L
INES OL WHERE OL.ORDER_NUMBER
= O.ORDER_NUMBER
```

Для построения результирующего множества Oracle должен обработать на четыре тысячи строк меньше. Может показаться, что это немного, но надо учитывать, что в загруженной системе Oracle должен будет сохранять согласованную по чтению копию обрабатываемых записей на всем протяжении запроса, и этих записей будет на 4000 меньше.

Еще одним вариантом может быть добавление выражения `MOD(order_number,2) = 0` непосредственно в текст запроса, что синтаксически

вполне допустимо. Однако оптимизатор Oracle может выбрать план выполнения, предусматривающий выборку всех позиций заказов, а удаление нечетных выполнить в оперативной памяти. Конечно, эту проблему можно решить использованием индекса по ключу-функции (function-based index), но этот путь тоже требует накладных расходов.

Переход к одному запросу также сводит на нет выгоду от использования массовой выборки позиций заказа в дальнейшем.

Oracle не обеспечивает для вложенных курсоров уровень изоляции `READ COMMITTED`. Результирующие множества поддерживаются только от момента неявного открытия и до последующего закрытия вложенного курсора. Однако для главного курсора обеспечен уровень изоляции `READ COMMITTED`.

Еще одной жизнеспособной альтернативой было бы использование двух курсоров, одного для выборки заказов, другого – для выборки позиций заказа. Но тогда оптимизатор вынужден будет рассматривать их как два отдельных курсора, так как ему неизвестно об их связи.

Заключение

В этой главе с двух разных точек зрения рассмотрено взаимодействие курсоров и PL/SQL. Первая связана с ежедневной работой по администрированию баз данных: такие механизмы, как повторное использование курсоров, их разбор и часто не замечаемый неполный разбор должны быть известны каждому администратору, так как влияют на работу практически каждого приложения Oracle. Вторая в большей степени ориентирована на разработку приложений; такие вопросы, как массовая выборка и использование типа `REF CURSOR`, хотя и не возникают в работе администратора базы данных ежедневно, но тоже требуют понимания. Ваша работа требует не только отслеживания и диагностики проблем с базой данных. Не менее важно дать правильные рекомендации разработчикам, чтобы ваша база данных не потеряла своей производительности.

3

Табличные функции

Табличной функцией называется функция, которая может быть использована в запросе в качестве источника данных. Например, вы можете поместить табличную функцию в предложение FROM оператора SELECT в программе на PL/SQL. Еще более важно, что табличная функция может возвращать записи. (Фактически она возвращает коллекцию объектов.) Уже эти две возможности делают табличные функции очень полезными в ситуациях, требующих сокрытия сложностей обработки за одним оператором SELECT, например в отчетах и компонентах API. Добавьте сюда возможности конвейерной и параллельной обработки, и вы получите мощнейший инструмент для работы с ETL-процессами (Extraction, Transformation, Loading – извлечение, преобразование и загрузка данных) в хранилищах данных.

Понятно, что табличные функции полезны для тех, кто разрабатывает отчеты и имеет дело с хранилищами данных, но у вас может возникнуть вопрос, зачем они нужны администраторам баз данных. В двух словах ответ таков: вам надо знать о них потому, что остальные сотрудники вашей организации могут быть не в курсе. Многие разработчики даже не слышали о существовании табличных функций, не говоря уже о том, чтобы использовать их для повышения производительности приложений – вот тут-то и нужен администратор базы данных. Представьте себе отчет, в котором запрос вслед за изменяющимися требованиями стал настолько сложным, что производительность стала неприемлемо низкой. Разработчик может перепробовать все комбинации подзапросов и внешних соединений – и все безрезультатно. Очевидно, что обработка стала намного сложнее того, что можно ожидать от одного оператора SELECT, но интерфейс отчета требует, чтобы она была сосредоточена в единственном операторе. В таких ситуациях администраторы баз данных часто бросаются в бой и стремятся помочь разработчику, строя планы выполнения и различным образом секциони-

руя таблицы в надежде достичь ускорения. Ничего не получается, вы в тупике и мечтаете о возможности сымитировать механизмы запроса Oracle, чтобы самостоятельно сформировать результирующее множество. Табличные функции позволяют это сделать!

В этой главе рассказывается, как работают табличные функции, и каким образом вы можете воспользоваться их преимуществами в рабочей среде. В ней также описываются механизм взаимодействия табличных функций с курсорами и способы конвейеризации, вложения и распараллеливания этих функций для достижения еще большей производительности. В завершение обсуждается использование табличных функций в ряде реальных приложений.

Зачем нужны табличные функции?

Начнем с простого примера, показывающего, как выглядят табличные функции и что они могут делать.

Простой пример

Выше уже высказывалась мысль о том, чтобы обращаться к табличной функции с помощью оператора SELECT. Приведем пример.

```
SELECT *
  FROM TABLE(company_balance_sheet);
```

На первый взгляд это выглядит как обычный запрос, но посмотрите внимательно: company_balance_sheet — это функция. Представим себе, что эта функция может анализировать миллионы пространных бухгалтерских записей о потенциальных приобретениях на предмет того, как они повлияют на итоговую прибыльность родительской компании. Огромный объем данных и строгие правила бухгалтерского учета требуют использования отдельной программы, но что делать, если результат должен быть доступен при помощи простого запроса, выданного с веб-страницы? На помощь приходят табличные функции.

Вот пример использования табличной функции в программе на PL/SQL. Заметьте, здесь она используется, как и любой другой курсор. Однако эта функция может просматривать множество детальных записей о транзакциях, вычисляя в реальном времени итоги по регионам, что дает возможность менеджерам принимать решения, связанные с планированием объемов продаж.

```
DECLARE
  CURSOR curs_get_western_sales IS
  SELECT *
    FROM TABLE(total_sales_by_region)
   WHERE region = 'Western';
  v_western_sales NUMBER;
BEGIN
  OPEN curs_get_western_sales;
```

```
    FETCH curs_western_sales INTO v_western_sales;
    CLOSE curs_get_western_sales;
  END;
```

Следующий пример показывает, что табличная функция способна принимать параметры, которые могут использоваться для управления процессом обработки данных. Здесь функция просматривает многочисленные результаты сложных анализов в поисках аномалий, прежде чем представить результаты, которые должны быть достаточно точными для оформления в реальном времени заявок на гранты, направляемые на финансирование важных исследований в области борьбы с раком:

```
SELECT *
  FROM TABLE(cancer_research_results( sdate => SYSDATE, edate => SYSDATE + 1 );
```

В чуть менее грандиозных (но не менее важных для вашего собственного бизнеса) масштабах вы можете использовать табличную функцию в запросе, использующем сложную логику для поиска в течение трех секунд всех заявок на внеплановую работу, чтобы в вызывающем приложении не произошел тайм-аут. Или вы можете использовать табличную функцию с целью исключить появление простого экрана с запросом в среде .NET – того, на котором выводятся данные о наличии запасных частей для неисправного водопровода, – так как сведения нельзя считать надежными в силу того, что расчет по формулам для их поиска выполнялся слишком долго.

В области преобразования данных табличные функции позволяют передавать результат промежуточного преобразования последовательно от одной функции к другой. Часто большие задачи по преобразованию данных вынуждены ждать сами себя из-за того, что последующие преобразования не могут начаться до окончания формирования полного результирующего множества начального преобразования – хотя логика программы позволяет им начать работу, как только станет доступна первая запись. В такой ситуации вы можете применить табличные функции в режиме, называемом *конвейеризацией*, также вы можете использовать *распараллеливание табличных функций* для достижения еще большего эффекта.

Вы еще не уверены, сделают ли табличные функции проще жизнь администратора базы данных? Тогда обратите внимание на тот факт, что табличные функции позволяют встраивать бизнес-логику в запросы, гарантируя тем самым, что эти запросы выполнят в точности те действия, для которых они предназначены. Например, вместо того, чтобы беспокоиться о размере сегмента отката для часто выполняющихся длинных запросов, вы, зная, что нужные данные обновляются с периодичностью раз в час, можете закэшировать их в памяти. Кроме того, в табличных функциях вы можете использовать такие механизмы повышения производительности, как массовая выборка и ассоциативные массивы (ранее называвшиеся индекс-таблицами), уменьшая общую нагрузку на базу данных.

Рассмотрению этих и других примеров посвящена данная глава. Для начала обратимся к основам.

Вызов табличной функции

Большинству администраторов БД известно, что Oracle позволяет вызывать функции в запросах, как в следующем примере:

```
SELECT SYSDATE
  FROM DUAL;
```

Oracle разрешает запросы такого типа, так как структура возвращаемого множества определена: в единственной записи будет возвращен единственный столбец типа DATE.

Большинству администраторов баз данных также известно, что структура результирующего множества должна быть определена для любого входящего в оператор SELECT объекта (т. е. таблицы, представления и др.). В противном случае база данных не сможет определить формат возвращаемых данных. Тем не менее непривычно, что функции, исторически возвращавшие единственное скалярное значение, могут возвращать результирующие множества, состоящие из нескольких записей и нескольких столбцов, например такие:

```
SQL> SELECT order_number,
  2      creation_date,
  3      assigned_date,
  4      closed_date
  5      FROM TABLE(order_history_function(region_id => 22))
  6      WHERE region = 11;

ORDER_NUMBER CREATION_DATE ASSIGNED_DATE CLOSED_DATE
------------ ------------- ------------- -----------
       10987     10-JAN-05     11-JAN-05   22-JAN-05
       10989     12-JAN-05     15-JAN-05   20-JAN-05
       10993     20-JAN-05     21-JAN-05   28-JAN-05
```

Определение структуры результирующего множества

Компилятор PL/SQL достаточно интеллектуален, чтобы определить тип результирующего множества для таблицы или представления. Он позволяет нам не беспокоиться о типах данных благодаря использованию атрибутов %TYPE и %ROWTYPE, как показано ниже.

```
DECLARE
  v_order_row orders%ROWTYPE;
BEGIN
  SELECT order_id,
         region_id
    INTO v_order_row
    FROM orders;
END;
```

Однако PL/SQL столкнется с трудностями, расшифровывая структуру возвращаемых табличной функцией данных, так как у него нет основы, на которую он мог бы опереться. Вы должны дать ему эту точку опоры явно, сославшись на объекты или коллекции Oracle. Сказанное проиллюстрировано в следующем примере, где объявлен объект, а затем коллекция таких объектов.

```
CREATE TYPE rowset_o AS OBJECT ( col1 NUMBER,
                                 col2 VARCHAR2(30));
/
CREATE TYPE rowset_t AS TABLE OF rowset_o;
/
```

Именно свойство коллекции иметь в себе много записей позволяет использовать ее в качестве результирующего множества функции.

```
CREATE OR REPLACE FUNCTION simple RETURN rowset_t AS
  v_rowset rowset_t := rowset_t();
BEGIN
  v_rowset.EXTEND(3);
  v_rowset(1) := rowset_o(1,'Value 1');
  v_rowset(2) := rowset_o(2,'Value 2');
  v_rowset(3) := rowset_o(3,'Value 3');
  RETURN(v_rowset);
END;
```

Все, что делает эта функция, – собирает три записи в коллекцию и возвращает их. Теперь функция может быть вызвана из оператора SELECT при помощи ключевого слова TABLE, сообщающего Oracle, что возвращаемую коллекцию следует интерпретировать как набор записей.

```
SQL> SELECT *
  2    FROM TABLE(simple);

      COL1 COL2
---------- ----------
         1 Value 1
         2 Value 2
         3 Value 3

3 rows selected.
```

К возвращаемому табличной функцией PL/SQL результирующему множеству может быть приложена вся мощь Oracle SQL, как если бы запрос выполнялся к таблице или представлению.

```
SQL> SELECT *
  2    FROM TABLE(simple)
  3   WHERE col1 = 2;

      COL1 COL2
---------- ----------
         2 Value 2
```

```
SQL> SELECT col2
  2    FROM TABLE(SIMPLE)
  3  GROUP BY col2;

COL2
------------------------
Value 1
Value 2
Value 3
```

Табличная функция может выполнять любые действия, доступные обычной функции, включая запросы и логические выражения.

Тема табличных функций довольно обширна, поэтому в этой главе мы не сможем обсудить в подробностях все возможные варианты их применения. Однако все упоминаемые здесь приложения подчиняются одному общему требованию: в них используется временное хранение данных в оперативной памяти (в коллекции). Рассмотрим, например, уже упоминавшееся приложение, применяющееся в исследовании рака. Каждая экспериментально полученная запись может рассматриваться на предмет включения в окончательный результат только после проверки на полноту; также необходимо проверить на полноту каждый отдельный эксперимент, прежде чем переходить к следующим шагам обработки. Если надо проверить 1000 экспериментов, это займет более получаса (2000 секунд), и только затем станет возможной дальнейшая обработка. Это приложение стало бы значительно эффективнее, если бы данные каждого эксперимента передавались для последующей обработки по мере прохождения проверки. Табличные функции, как показано в следующем разделе, предоставляют такую возможность.

Курсоры, конвейеризация, вложение

К этому моменту у вас, наверное, сложилось впечатление, что табличные функции – это универсальное средство повышения производительности, позволяющее решить любую проблему. Однако рассматриваемые далее возможности: применение курсоров, конвейеризация и вложение табличных функций – настолько эффективны, что могут побудить вас даже переписать код, лишь бы их использовать.

Курсоры

Вездесущий курсор появляется в табличной функции в двух ролях: как тип данных параметра и как SQL-функция, позволяя передавать оператор `SELECT` для выполнения непосредственно в табличную функцию.

Конвейеризация

Данная особенность позволяет табличной функции возвращать строки результирующего множества по одной, не накапливая его целиком. Благодаря этому последующая обработка может начаться гораздо раньше. Возвращаясь к уже упоминавшемуся примеру с рако-

выми исследованиями, посмотрим, что произойдет, если функции надо выполнить 100 проверок, каждая из которых занимает 3 секунды. Это означает, что следующий этап обработки сможет начаться только через 5 минут. При конвейерной обработке он начнется через 3 секунды.

Вложение

Для выполнения нескольких операций над данными можно использовать вложение табличных функций. Этот способ особенно полезен при работе с ETL-процессами хранилищ данных.

Вероятно, лучший способ продемонстрировать перечисленные возможности – это привести пример ETL-процесса в хранилище данных, извлекающего информацию о нарядах на работу. Рассмотрим функцию, извлекающую сведения о дате создания, выдачи и закрытия нарядов. Затем эти компоненты передаются для дальнейшей обработки процессу ETL.

Курсоры

Мы уже рассматривали подробно курсоры в главе 2, и вот опять они здесь! Как взаимодействуют курсоры и табличные функции? Для начала рассмотрим табличную функцию без конвейеризации.

```
/* Файл на веб-сайте: date_parser.sql */
CREATE OR REPLACE FUNCTION date_parse ( p_curs SYS_REFCURSOR )
                  RETURN order_date_t AS

  v_order_rec orders%ROWTYPE;
  v_ret_val   order_date_t := order_date_t();

BEGIN

  -- для каждого заказа в курсоре...
  LOOP
    FETCH p_curs INTO v_order_rec;
    EXIT WHEN p_curs%NOTFOUND;

    -- добавить в массив 3 элемента и записать туда даты
    -- создания, выдачи и закрытия наряда
    v_ret_val.EXTEND(3);
    v_ret_val(v_ret_val.LAST - 2) := order_date_o(v_order_rec.order_number,
                                                  'O',
                                     TO_CHAR(v_order_rec.create_date,'YYYY'),
                                     TO_CHAR(v_order_rec.create_date,'Q'),
                                     TO_CHAR(v_order_rec.create_date,'MM'));
    v_ret_val(v_ret_val.LAST - 1) := order_date_o(v_order_rec.order_number,
                                                  'A',
                                     TO_CHAR(v_order_rec.assign_date,'YYYY'),
                                     TO_CHAR(v_order_rec.assign_date,'Q'),
                                     TO_CHAR(v_order_rec.assign_date,'MM'));
    v_ret_val(v_ret_val.LAST) := order_date_o(v_order_rec.order_number,
                                              'C',
```

```
                                    TO_CHAR(v_order_rec.close_date,'YYYY'),
                                    TO_CHAR(v_order_rec.close_date,'Q'),
                                    TO_CHAR(v_order_rec.close_date,'MM'));
  END LOOP;  -- для каждого заказа в курсоре

  RETURN(v_ret_val);

END;
```

Вот результат запроса для трех нарядов.

```
ORDER_NUMBER D       YEAR    QUARTER      MONTH
------------ - ---------- ---------- ----------
           1 O       2005          3          8
           1 A       2005          3          8
           1 C       2005          3          8
           2 O       2005          4         10
           2 A       2005          4         10
           2 C       2005          4         10
           3 O       2005          4         12
           3 A       2005          4         12
           3 C       2005          4         12
```

Заметьте, я сказал, что в запросе было три наряда, но предусмотрительно не сообщил, откуда выполнялась выборка. Дело в том, что выборка осуществляется из переданного функции курсора, имеющего такой вид:

```
SELECT *
  FROM TABLE(date_parse(CURSOR(SELECT *
                               FROM orders)));
```

Ключевое слово `TABLE` означает, что выполняется вызов функции `date_parse`. Ключевое слово `CURSOR` означает, что следующий за ним текст (в скобках) должен использоваться как курсор в табличной функции. Он неявно открывается, и функция извлекает из него записи. Когда табличная функция выходит из области видимости, он неявно закрывается. Способность передавать в функцию текст курсора – очень мощный инструмент, так как требует всего лишь корректного SQL. Функция может быть вызвана (и, следовательно, повторно использована) в разных местах. Например, если надо обработать только сегодняшние записи, то вызов должен быть таким:

```
SELECT *
  FROM TABLE(date_parse(CURSOR(SELECT *
                               FROM orders
                              WHERE create > TRUNC(SYSDATE))));
```

Далее, если надо ограничить выборку определенными регионами, добавим следующее ограничение в оператор `SELECT`:

```
SELECT *
  FROM TABLE(date_parse(CURSOR(SELECT *
                               FROM orders
```

```
                                    WHERE create > TRUNC(SYSDATE)
                                      AND region_id = 33)));
```

Единственное требование к этому оператору SELECT заключается в том, что должны выбираться все столбцы таблицы ORDERS, так как в табличной функции переменная типа запись, в которую помещается результат, объявлена как orders%ROWTYPE. Однако данное требование не всегда обязательно, есть много способов установить соответствие между операторами SELECT, типами параметров-курсоров и локальными переменными. Мы рассмотрим их далее в этой главе.

Конвейеризованные табличные функции

Конвейеризованная табличная функция – это функция, которая возвращает результирующее множество в виде коллекции, но делает это итеративно. Другими словами, Oracle не ждет, когда выполнение функции закончится, накапливая все полученные строки в коллекции PL/SQL, прежде чем вернуть их. Вместо этого записи по мере их готовности к включению в коллекцию «выкачиваются» из функции. Давайте посмотрим на конвейеризованную табличную функцию в действии.

```
/* Файл на веб-сайте: date_parser_pipelined.sql */
CREATE OR REPLACE FUNCTION date_parse ( p_curs SYS_REFCURSOR )
                 RETURN order_date_t
                 PIPELINED AS

  v_order_rec orders%ROWTYPE;

BEGIN

  -- для каждого заказа в курсоре...
  LOOP
    FETCH p_curs INTO v_order_rec;
    EXIT WHEN p_curs%NOTFOUND;

    -- передать составляющие даты создания наряда
    PIPE ROW(order_date_o(v_order_rec.order_number,
                          'O',
                          TO_CHAR(v_order_rec.create_date,'YYYY'),
                          TO_CHAR(v_order_rec.create_date,'Q'),
                          TO_CHAR(v_order_rec.create_date,'MM')));

    -- передать составляющие даты выдачи наряда
    PIPE ROW(order_date_o(v_order_rec.order_number,
                         'A',
                         TO_CHAR(v_order_rec.assign_date,'YYYY'),
                         TO_CHAR(v_order_rec.assign_date,'Q'),
                         TO_CHAR(v_order_rec.assign_date,'MM')));

    -- передать составляющие даты закрытия наряда
    PIPE ROW(order_date_o(v_order_rec.order_number,
                         'C',
                         TO_CHAR(v_order_rec.close_date,'YYYY'),
```

```
                              TO_CHAR(v_order_rec.close_date,'Q'),
                              TO_CHAR(v_order_rec.close_date,'MM')));
  END LOOP;  -- для каждого заказа в курсоре
  RETURN;
END;
```

Между неконвейеризованной и конвейеризованной версиями имеются четыре синтаксических различия:

- Ключевое слово `PIPELINED` добавляется в заголовок функции с целью сообщить Oracle о необходимости возвращать результат немедленно, а не накапливать предварительно все результирующее множество.
- Команда `PIPE ROW` обозначает место, в котором функция возвращает отдельную запись.
- Одинокое ключевое слово `RETURN` осталось, но не делает ничего, осуществляя лишь выход из функции. Все результаты уже были переданы по конвейеру командой `PIPE ROW`.
- Возвращаемый тип данных (`order_date_o`) отличается от типа, указанного в объявлении функции (`order_date_t`). Несмотря на такое синтаксическое несоответствие, эти типы должны быть связаны между собой, как объясняется в следующем абзаце.

Oracle не позволит вернуть из произвольной табличной функции любой из обычных типов данных. Указанная в качестве возвращаемого типа функции коллекция должна иметь в качестве элемента объектный тип. В нашем примере возвращаемые типы данных были объявлены так:

```
CREATE OR REPLACE TYPE order_date_o AS OBJECT ( order_number NUMBER,
                                                date_type    VARCHAR2(1),
                                                year         NUMBER,
                                                quarter      NUMBER,
                                                month        NUMBER );
/
CREATE TYPE order_date_t AS TABLE OF order_date_o;
/
```

Чтобы показать, как много дает конвейеризация в такой ситуации, увеличим количество нарядов до 10 000 и выполним следующий запрос с использованием обеих версий (с конвейеризацией и без) нашей функции.

```
SELECT *
  FROM TABLE(date_parse(CURSOR(SELECT *
                               FROM orders)))
 WHERE ROWNUM <= 10;
```

Такой запрос отлично подходит для нашей цели, так как точно показывает, сколько времени необходимо функции для того, чтобы вернуть десятую по счету запись. Победителем становится...

- Второе место с достойным результатом в 2,73 секунды занимает версия без конвейеризации.
- Первое место с невероятным результатом в 0,07 секунды достается конвейеризованной версии.

В конвейеризованном варианте запрос выполняется быстрее на 2,66 секунды или на 93%. Более существенным с точки зрения администратора БД является то, что база данных тратит при выполнении запроса на 93% меньше времени на поддержание согласованной по чтению копии таблицы ORDERS, а это на 93% уменьшает вероятность возникновения избыточных операций физического чтения, вызванных продолжительностью запроса. Можно продолжать еще и еще.

Еще более приятно для администратора то, что при выполнении конвейеризованной версии расходуется меньше оперативной памяти, выделяемой Oracle сеансу. Здесь, для простоты, мы ограничим концепцию памяти сеанса понятиями областей UGA (User Global Area – глобальная область пользователей) и PGA (Program Global Area – программная глобальная область). В табл. 3.1 приведено сравнение объемов UGA и PGA, расходуемых при выполнении обеих версий функции. Эти данные получены после входа в систему и однократного выполнения конвейеризованной и неконвейеризованной функций.

Таблица 3.1. Затраты памяти сеанса Oracle на выполнение конвейеризованной и неконвейеризованной функций

	Неконвейеризованная	**Конвейеризованная**	**Разность**
Максимально в UGA	7105168	90284	7014884
Максимально в PGA	12815736	242708	12573028

В обеих областях (UGA и PGA) сокращение составляет 98% – это сокращает нагрузку на базу данных и позволяет ей работать быстрее.

Вложенные табличные функции

Вложением табличных функций называется такой способ их выполнения, при котором результат работы первой из них передается для дальнейшей обработки в следующую, затем ее результат – в следующую функцию и т. д. Такой способ иногда называют «гирляндой», или «цепочкой» (daisy-chaining). В сочетании с конвейеризацией вложение табличных функций образует исключительно мощный механизм для использования в ETL-процессах.

Покажем это на примере функции, принимающей результаты выполнения нашей табличной функции date_parse посредством курсора и выполняющей действия над ними. Чтобы не запутывать дело сложными ETL-преобразованиями, ограничимся простым сложением полученных значений order_number, year, quarter и month. Вот эта функция.

```
/* Файл на веб-сайте: next_in_line.sql */
CREATE OR REPLACE FUNCTION next_in_line ( p_curs SYS_REFCURSOR )
                  RETURN next_t
                  PIPELINED IS

  v_ret_val next_t := next_t();

  -- локальные переменные для полей курсора
  v_on NUMBER;
  v_dt VARCHAR2(1);
  v_yr NUMBER;
  v_qt NUMBER;
  v_mt NUMBER;

BEGIN

  -- для всех компонентов даты, полученных курсором...
  LOOP

    FETCH p_curs INTO v_on, v_dt, v_yr, v_qt, v_mt;
    EXIT WHEN p_curs%NOTFOUND;
    -- передать сумму компонентов
    PIPE ROW(next_o(v_on + v_yr + v_qt + v_mt));

  END LOOP;  -- для каждого компонента даты

  RETURN;

END;
```

Большинство синтаксических конструкций этого примера мы уже рассматривали. Новым является только синтаксис, используемый для вложения функций:

```
SELECT *
  FROM TABLE(next_in_line
             (CURSOR
              (SELECT *
                 FROM TABLE(date_parse
                            (CURSOR
                             (SELECT *
                                FROM orders))))));
```

Благодаря удачному форматированию они даже выглядят вложенными, это видно по расположению ключевых слов `TABLE` и `CURSOR`. В двух словах этот пример делает следующее: вызывает функцию `date_parse`, которая передает результаты функции `next_in_line`, которая передает их во внешний мир.

Распараллеливание табличных функций

Включение функций в парадигму оператора `SELECT` позволяет воспользоваться преимуществами еще одной особенности Oracle – параллельного выполнения запросов.

В Oracle уже давно обеспечен параллелизм как средство работы с большими запросами, позволяющее распределить обработку между несколькими PQ-серверами (Parallel Query – параллельные запросы), каждый из которых рассчитывает свою часть результата; затем данные собираются в единое результирующее множество. Внутренние механизмы Oracle определяют, как распределить работу между имеющимися PQ-серверами, чтобы получить наилучший результат. Администратор базы данных может влиять на процесс принятия решения, устанавливая степени параллелизма для таблиц или создавая специальные схемы секционирования, но, в конечном счете, наилучший способ выполнения запроса выбирает сервер Oracle.

Преимущества параллельных запросов

Распараллеливание полезно для любых типов запросов, независимо от того, используют ли они табличные функции. Запросы без табличных функций получают преимущество от ускоренного выполнения параллельных запросов, позволяющего быстрее собрать результирующее множество. Запросы, использующие табличные функции, выигрывают в оперативности не только от использования PQ-серверов, но и оттого, что формируют конечный результат по ходу работы.

Давайте проиллюстрируем это на примере таблицы, содержащей по одной записи на каждую транзакцию, выполняющуюся в большом банке:

```
SQL> DESC acct_transactions
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------
 AREA                                               VARCHAR2(10)
 TRX_DATE                                           DATE
 TRX_AMT                                            NUMBER
```

Предположим, нам надо разработать функцию, суммирующую транзакции по регионам. Сначала надо создать запрос с группировкой по регионам. Это осложняется необходимостью выполнения ряда сложных проверок; для наглядности в приводимых примерах соберем эти проверки в функцию `super_complex_validation`.

Вот искомый результат, полученный на демонстрационном наборе данных в предположении, что все транзакции прошли эту сложную проверку.

```
SQL> SELECT area,
  2         SUM(trx_amt)
  3    FROM acct_transactions
  4  GROUP BY area;

AREA       SUM(TRX_AMT)
---------- ------------
1                   460
```

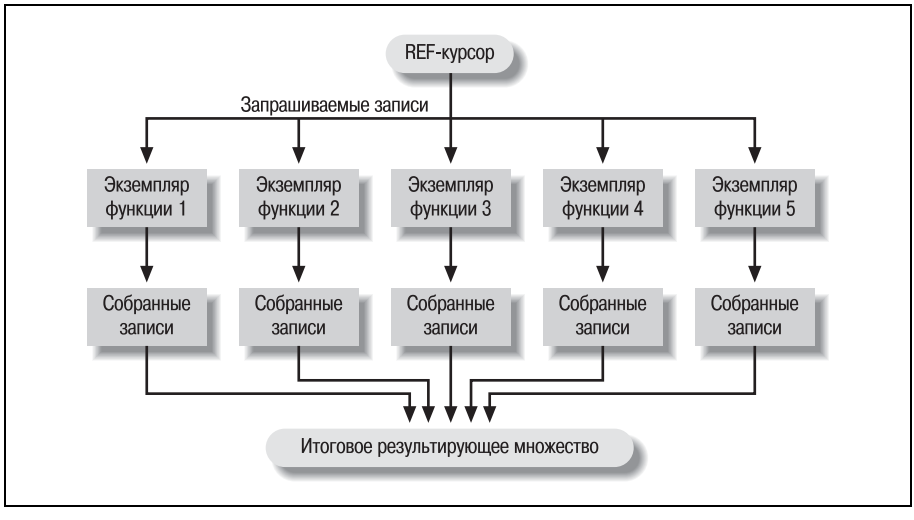


Рис. 3.1. *Распараллеленная табличная функция*

10	550
2	470
3	480
4	490
5	500
6	510
7	520
8	530
9	540

Применим параллельную обработку, чтобы получить этот результат быстрее при большом количестве записей. В случае использования табличных функций это означает, что Oracle запустит одновременно несколько экземпляров табличной функции и распределит между ними результаты, полученные от переданного REF-курсора, как показано на рис. 3.1.

Распределение записей

Одна из замечательных особенностей параллельных табличных функций заключается в том, что вы можете дать указания Oracle, как следует распределять записи между параллельными экземплярами функции. Можно указать две различные характеристики, влияющие на распределение. Это, во-первых, *секционирование* (*partitioning*), на основании которого Oracle решает, какие записи какому экземпляру функции направить. И, во-вторых, *организация потока* (*streaming*), определяющая порядок вывода выделенных экземпляру функции строк. Ниже для каждой из характеристик перечислены возможные варианты.

Секционирование

Записи могут быть разделены по диапазонам значений одного или нескольких столбцов, по рассчитанным для одного или нескольких столбцов хеш-значениям или просто по усмотрению Oracle. Предложение PARTITION BY указывает определенный способ секционирования для табличной функции, как показано в последующих примерах.

Организация потока

Записи могут быть упорядочены или сгруппированы по значениям определенных столбцов. В нижеследующих примерах будет показано использование ключевых слов ORDER и CLUSTER для организации выходного потока табличной функции.

Назначение и взаимодействие всех этих компонентов мы рассмотрим на примере разработки функции подсчета транзакций по учетным записям.

Произвольное секционирование (PARTITION BY ANY)

Для начала создадим функцию и заставим сервер Oracle распараллелить ее, оставив секционирование записей на его усмотрение. Предложение PARALLEL_ENABLE в заголовке функции сообщает Oracle, что код этой функции написан в расчете на параллельное выполнение, и использование этой возможности весьма желательно. Параметр PARTITION BY ANY в этом примере указывает, что записи, возвращаемые курсором типа REF CURSOR, должны быть секционированы случайным образом в том порядке, какой Oracle сочтет наиболее подходящим для имеющихся PQ-серверов. Другими словами, в данном примере PQ-серверы используются только для повышения производительности.

```
/* Файл на веб-сайте: pipelined.sql */
CREATE OR REPLACE FUNCTION area_summary ( p_cursor SYS_REFCURSOR )
                          RETURN area_summary_t
                          PIPELINED
                          PARALLEL_ENABLE ( PARTITION p_cursor BY ANY ) AS

  v_row   acct_transactions%ROWTYPE;
  v_total NUMBER := NULL;
  v_area  acct_transactions.area%TYPE;

BEGIN

  -- для каждой транзакции
  FETCH p_cursor INTO v_row;
  LOOP
    EXIT WHEN p_cursor%NOTFOUND;

    -- если прошли подробную проверку корректности
    IF super_complex_validation(v_row.trx_date,v_row.trx_amt) THEN

      -- инициализировать итог или увеличить итог текущего региона
      -- или вернуть итог по региону
```

```
      IF v_total IS NULL THEN
        v_total := v_row.trx_amt;
        v_area := v_row.area;
      ELSIF v_row.area = v_area THEN
          v_total := v_total + v_row.trx_amt;
      ELSE
        PIPE ROW(area_summary_o(v_area,v_total));
        v_total := v_row.trx_amt;
        v_area := v_row.area;
      END IF;

    END IF;  -- подробная проверка

    FETCH p_cursor INTO v_row;

  END LOOP; -- для каждой транзакции

  PIPE ROW(area_summary_o(v_area,v_total));

END;
```

При выполнении этой функции используется оператор SELECT, получающий в качестве параметра оператор SELECT. Не могу не повторить еще раз!

```
SELECT *
  FROM TABLE(area_summary(CURSOR(SELECT *
                                   FROM acct_transactions)));
```

Результат будет случайным и непредсказуемым, так как функция рассчитывает на получение записей, отсортированных по региону, но фактически распределение записей оставлено на усмотрение Oracle. Поэтому параметр PARTITION BY ANY не поможет этой функции. (Хотя позволяет получить неверный результат очень быстро!)

Секционирование по диапазону (PARTITION BY RANGE)

Для полного использования преимуществ параллельной обработки надо заставить Oracle отправлять все записи одного региона одному и тому же экземпляру функции. Например, если параллельно выполняются три экземпляра функции, то все записи региона 7 должны попасть в экземпляр 1, 2 или 3. Такой способ указывается с помощью предложения RANGE, как показано в новом заголовке функции:

```
CREATE OR REPLACE FUNCTION area_summary ( p_cursor ref_cursors.acct_trx_curs )
                        RETURN area_summary_t
                        PIPELINED
                       PARALLEL_ENABLE ( PARTITION p_cursor BY RANGE(area) ) AS
```

Обратите внимание, что здесь для параметра REF CURSOR использована строгая типизация. То есть Oracle для корректного разделения записей должен заранее знать их структуру. Наш пакет ref_cursors содержит единственную строку:

```
PACKAGE ref_cursors IS
  TYPE acct_trx_curs IS REF CURSOR RETURN acct_transactions%ROWTYPE;
END;
```

Теперь результат должен стать лучше.

```
SQL> SELECT *
  2    FROM TABLE(area_summary(CURSOR(SELECT *
  3                                     FROM acct_transactions)));

AREA              AMT
---------- ----------
6                  96
7                  97
8                  98
9                  99
6                 414
7                 423
8                 432
9                 441
1                 369
10                450
2                 378
3                 387
4                 396
5                 405
1                  91
2                  92
3                  93
4                  94
5                  95
10                100

20 rows selected.
```

Упорядочение потока (ORDER)

Получилось пока не совсем то, что нужно. Проблема в том, что хотя записи распределяются между экземплярами соответственно региону, они не сортируются в пределах каждого экземпляра функции. Например, все записи 6-го региона отправляются одному экземпляру функции, но туда же попадают записи других регионов, что приводит к ошибкам в расчете итогов по региону 6. Надо сообщить Oracle, что записи должны сортироваться по региону еще и в каждом из экземпляров функции. Это делается с помощью предложения ORDER, как показано ниже.

```
CREATE OR REPLACE FUNCTION area_summary ( p_cursor ref_cursors.acct_trx_curs )
                         RETURN area_summary_t
                         PIPELINED
                        PARALLEL_ENABLE ( PARTITION p_cursor BY RANGE(area) )
                          ORDER p_cursor BY (area) AS
```

Посмотрим, что получилось.

```
SQL> SELECT *
  2    FROM TABLE(area_summary(CURSOR(SELECT *
  3                                     FROM acct_transactions)));

AREA              AMT
---------- ----------
1                 460
10                550
2                 470
3                 480
4                 490
5                 500
6                 510
7                 520
8                 530
9                 540

10 rows selected.
```

Есть успех! Теперь итоговые значения верны, так как полученные курсором записи были разделены между параллельными экземплярами функции в соответствии с регионом *и* отсортированы в каждом из них по региону, как показано на рис. 3.2.

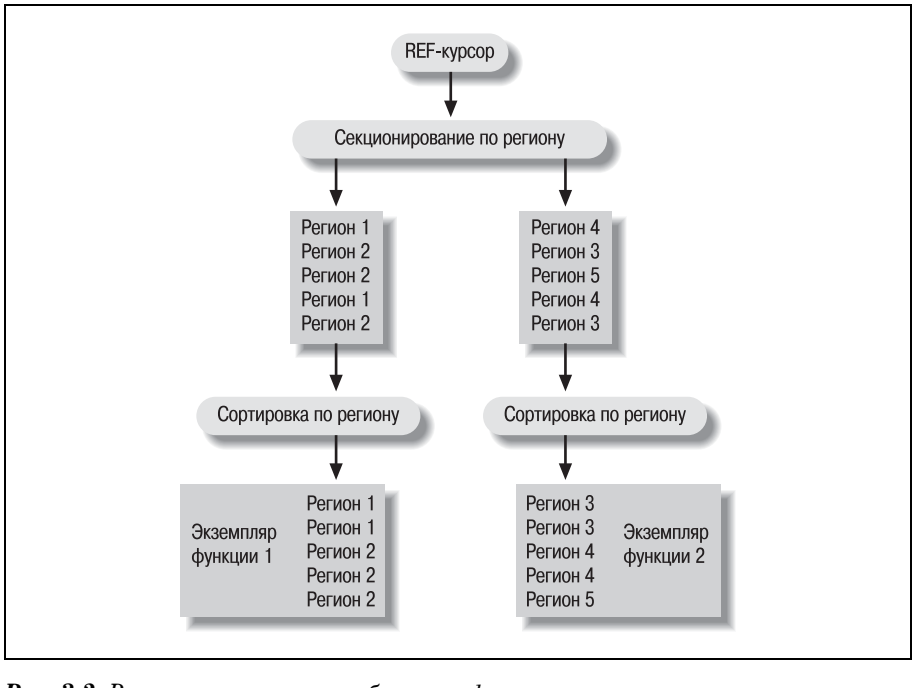


Рис. 3.2. *Распараллеленная табличная функция, секционированная и отсортированная по региону*

Хеш-секционирование (PARTITION BY HASH)

В предыдущих примерах показано произвольное (ANY) секционирование и секционирование по диапазону (RANGE). Возможен еще один вариант, называемый хеш-секционированием, при котором для заданных столбцов рассчитывается хеш-значение, и на основании этого значения выбирается экземпляр функции, которому будет передана данная запись. Совпадающие значения дадут одинаковые значения хеша, поэтому такое распределение записей будет правильно работать в случае нашей функции подсчета транзакций по учетным записям. На самом деле вариант с параметром PARTITION BY HASH обычно работает чуть быстрее, чем с PARTITION BY RANGE.

Вот его синтаксис:

```
CREATE OR REPLACE FUNCTION area_summary ( p_cursor ref_cursors.acct_trx_curs )
                          RETURN area_summary_t
                          PIPELINED
                           PARALLEL_ENABLE ( PARTITION p_cursor BY HASH(area) )
                          ORDER p_cursor BY (area) AS
```

Группировка потока (CLUSTER)

В предыдущей версии нашей суммирующей функции мы использовали предложение ORDER, так как в этой функции предполагается, что записи для каждого из ее экземпляров упорядочены по региону. Еще один параметр, CLUSTER, гарантирует, что записи с одинаковыми значениями в указанных столбцах будут сгруппированы в пределах экземпляра функции. Однако помните, что этот параметр не вызывает сортировки значений.

В силу того, что в нашей функции рассматривается только один столбец, можно легко перейти к группировке. Для этого используется следующий синтаксис:

```
CREATE OR REPLACE FUNCTION area_summary ( p_cursor ref_cursors.acct_trx_curs )
                      RETURN area_summary_t
                      PIPELINED
                      PARALLEL_ENABLE ( PARTITION p_cursor BY HASH(area) )
                      CLUSTER p_cursor BY (area) AS
```

Проверка показывает, что вариант с параметром CLUSTER для одного столбца работает быстрее варианта с параметром ORDER.

Какой вариант выбрать?

Какой из всех этих вариантов секционирования и организации потока лучше подходит в вашем конкретном случае?

PARTITION BY ANY (произвольное секционирование)

Используйте этот вариант, если вам нужна максимальная скорость *и* порядок следования записей вообще не важен. Например, если

вам надо выполнить независимые вычисления для каждой записи курсора REF CURSOR, выберите данный способ, так как Oracle распределит записи, не заботясь об их порядке.

Мне еще не встречались ситуации, когда требовалось бы сочетание произвольного секционирования с параметрами ORDER или CLUSTER, но такая комбинация синтаксически допустима.

PARTITION BY RANGE (секционирование по диапазону)

Указывайте этот параметр, если ваша функция предусматривает совместную обработку определенных записей *и* эти записи равномерно распределены по значениям секционирования. В этом случае все экземпляры параллельной функции будут выполнять примерно равные объемы работы.

Этот тип секционирования можно сочетать с параметрами ORDER BY или CLUSTER BY. Имейте в виду, что вариант с группировкой будет работать немного быстрее.

PARTITION BY HASH (хеш-секционирование)

Используйте этот параметр, если ваша функция требует, чтобы определенные записи обрабатывались совместно, *и* распределение имеет некоторый перекос. Алгоритм хеширования увеличивает шансы того, что все экземпляры параллельной функции будут выполнять примерно равные объемы работы.

Допустимо комбинирование данного параметра с ORDER BY или CLUSTER BY. При этом вариант с параметром CLUSTER работает чуть быстрее.

Что делает Oracle?

У вас, наверное, уже возник вопрос, как база данных выполняет секционирование результатов запроса. Выполняет ли Oracle по одному запросу в каждом экземпляре параллельной функции или он выполняет единственный запрос и секционирует его результат? Давайте выясним это, обратившись к разделяемому пулу.

```
SQL> SELECT sql_text,
  2         parse_calls,
  3         executions
  4    FROM v$sql
  5   WHERE INSTR(UPPER(sql_text),'ACCT_TRANSACTIONS')  > 0
  6     AND INSTR(UPPER(sql_text),'SQL_TEXT') = 0
  7     AND command_type = 3;

SQL_TEXT                  PARSE_CALLS EXECUTIONS
------------------------- ----------- ----------
SELECT *   FROM TABLE(are           1          1
a_summary(CURSOR(SELECT *
FROM acct_transactions)))

SELECT "A3"."AREA" "AREA"           1          1
,"A3"."TRX_DATE" "TRX_DAT
```

```
E",'A3'."TRX_AMT" "TRX_AM
T" FROM "ACCT_TRANSACTION
S" "A3" ORDER BY "A3"."AR
EA"
```

Здесь присутствуют только два курсора: один, выполненный нами, и один, выполненный в табличной функции; у обоих счетчики разборов и выполнений содержат 1. Это означает, что используется единственный курсор, а Oracle по необходимости выполняет секционирование полученных строк.

Сколько нужно PQ-серверов?

Для большинства операций с параллельными запросами можно ограничить количество (или хотя бы повлиять на него) задействованных в них PQ-серверов. Например, можно задать степень параллелизма для таблицы, и Oracle будет стремиться обеспечить требуемое количество PQ-серверов для запросов к этой таблице. Также можно задать степень параллелизма (а, следовательно, и количество используемых PQ-серверов) для любого конкретного запроса, указав соответствующие подсказки оптимизатору.

К сожалению, такие возможности отсутствуют для параллельных табличных функций.

Хотелось бы иметь возможность использовать синтаксис такого типа:

```
CREATE OR REPLACE FUNCTION area_summary ( p_cursor ref_cursors.acct_trx_curs )
                          RETURN area_summary_t
                          PIPELINED
                           PARALLEL_ENABLE ( PARTITION p_cursor BY HASH(area) )
                          DEGREE 3 - invalid - DRH dream syntax
                          ORDER p_cursor BY (area) AS...
```

Так можно было бы задать максимальное количество PQ-серверов, используемых при одном обращении к табличной функции. Несмотря на эту легкую критику, надо признать, что Oracle достаточно хорошо справляется с задачей определения требуемого количества PQ-серверов.

Использование табличных функций

В этом разделе мы на реальном примере из практики большой кабельной компании, пытающейся отслеживать повторные наряды, покажем, как используются табличные функции. Говоря попросту, повторный наряд – это направление техника по одному и тому же адресу для повторного выполнения (или исправления) уже сделанной работы в течение 30 дней. Повторный визит не всегда связан с наличием претензий – он может быть результатом установки оборудования в 30-дневный период после предварительного заказа.

Адреса определяются предопределенными идентификаторами. Они могут соответствовать чему угодно, от конкретной кабельной розетки

до отдельного дома или большого торгового центра. Вид работы определяется идентификатором типа наряда. Например, тип 1 может означать «Прокладка кабеля», а тип 2 – «Замена кабеля».

Компания имеет отделения в нескольких регионах, в каждом из которых используется собственный набор идентификаторов адресов, типов нарядов и критериев повторного наряда. Эти критерии определяют пары типов нарядов, которые должны быть выданы по одному адресу в течение 30-дневного периода, чтобы наряд был признан повторным. Типы нарядов в такой паре могут совпадать или не совпадать, например «Ремонт кабеля», произошедший после «Прокладки кабеля», может считаться таким же повторным нарядом, как и два последовательных наряда на «Прокладку кабеля».

Эти критерии хранятся в таблице:

```
SQL> DESC repeat_order_criteria
 Name                                      Null?    Type
 ----------------------------------------- -------- ------
 REGION_ID                                          NUMBER
 START_DATE                                         DATE
 FIRST_TYPE_ID                                      NUMBER
 REPEAT_TYPE_ID                                     NUMBER
```

Эта таблица содержит определения повторных нарядов для каждого региона, включая дату их ввода в действие. Вот пример записи.

```
SQL> SELECT *
  2    FROM repeat_order_criteria;

 REGION_ID START_DAT FIRST_TYPE_ID REPEAT_TYPE_ID
---------- --------- ------------- --------------
         1 19-APR-05            44            102
```

В этой записи говорится, что в регионе 1 наряд с типом 102, следующий в 30-дневный период за нарядом 44 и по тому же адресу, считается повторным.

Таблица ORDERS содержит поля, необходимые для определения «повторности» наряда.

```
SQL> DESC orders
 Name                                      Null?    Type
 ----------------------------------------- -------- ------
 ORDER_NUMBER                              NOT NULL NUMBER
 ORDER_DATE                                NOT NULL DATE
 REGION_ID                                 NOT NULL NUMBER
 TYPE_ID                                   NOT NULL NUMBER
 LOCATION_ID                               NOT NULL NUMBER
```

Дополнительную сложность этим условиям придает то, что компания обрабатывает ежедневно десятки тысяч заявок, а также то, что критерии повторного наряда могут измениться в любой момент, поэтому новый набор данных должен быть получен быстро. Большое количество

записей в сочетании с необходимостью «поштучной» обработки записей для получения результирующего множества делает это приложение идеальным кандидатом на применение табличных функций.

Заголовок функции

Для начала разберемся, из чего состоит заголовок нашей функции.

```
1 CREATE FUNCTION repeat_order_finder ( p_curs cursors.repeat_orders_curs )
2                RETURN repeat_region_location_t
3                PIPELINED
4                PARALLEL_ENABLE ( PARTITION p_curs BY RANGE(region_id) )
5                ORDER p_curs BY (location_id, order_date) IS
```

Как это свойственно заголовкам функций, здесь можно много о чем рассказать, поэтому разберем его строка за строкой.

Строка 1

Определяет имя функции и ее параметр – курсор типа `REF CURSOR` со строгим контролем типа, объявленный в другом пакете, например, так:

```
CREATE OR REPLACE PACKAGE cursors AS
  TYPE repeat_orders_rec IS RECORD (order_number NUMBER,
                                    order_date   DATE,
                                    region_id    NUMBER,
                                    type_id      NUMBER,
                                    location_id  NUMBER );

  TYPE repeat_orders_curs IS REF CURSOR RETURN repeat_orders_rec;
END;
```

При обращении к функции будем передавать ей оператор `SELECT`, возвращающий все наряды за последние 30 дней.

Строка 2

Определяет структуру возвращаемых функцией записей. Данный тип был создан с помощью следующих SQL-определений объекта и коллекции:

```
CREATE TYPE repeat_region_location_o AS OBJECT ( region_id      NUMBER,
                                                 location_id    NUMBER,
                                                 first_type_id  NUMBER,
                                                 repeat_type_id NUMBER );
/
CREATE TYPE repeat_region_location_t AS TABLE OF
repeat_region_location_o;
/
```

Строка 3

Указывает на то, что функция конвейеризирует возвращаемые строки по мере их появления.

Строка 4

Определяет, каким способом будут распределяться по нескольким параллельным экземплярам функции строки, полученные переданным ей курсором. Строки должны разделяться по значению столбца REGION_ID. Это значит, что все записи определенного региона будут обработаны одним экземпляром функции. Из этого *не следует*, что будет создано по одному экземпляру для каждого региона. Oracle по своему усмотрению распределит имеющиеся PQ-серверы для выполнения экземпляров функции. Поэтому один экземпляр может обрабатывать несколько регионов.

Строка 5

Указывает, что в каждом из экземпляров записи должны быть отсортированы по столбцам LOCATION_ID и ORDER_DATE.

Простой разбор заголовка показал большинство возможностей, скрытых в табличной функции. Эта функция позволяет использовать мощь параллельных вычислений для последовательной обработки десятков тысяч записей. Более того, она позволяет указать, какие записи будут переданы каждому из экземпляров параллельной функции, благодаря чему при кодировании можно использовать некоторые допущения. Добавим к этому отсутствие необходимости дожидаться окончания обработки всех записей, чтобы вернуть результат, и – вот она, нирвана!

Но хватит проповедей! Вернемся к нашей функции.

Основной цикл

Основной алгоритм нашей функции заключается в циклической выборке записей из REF CURSOR, как показано в этом псевдокоде.

```
BEGIN

  -- для каждого наряда...
  LOOP

    FETCH p_curs INTO v_order;
    EXIT WHEN p_curs%NOTFOUND;

    IF повторный_наряд THEN
      PIPE ROW();
    END IF;

  END LOOP;  -- для каждого наряда

  RETURN;

END;
```

Здесь все просто. Всего лишь извлекаем записи из переданного в функцию курсора и проверяем их на соответствие критериям повторного наряда. Если условие выполнено, отправляем запись в конвейер. Теперь добавим загрузку критериев для данного региона.

Массовая выборка критериев

Благодаря тому, что наша табличная функция гарантирует, что записи будут сгруппированы по регионам, можно быть уверенным, что при каждом изменении идентификатора региона в извлекаемой записи происходит переход к следующему региону. А первое, что надо сделать для нового региона, – это загрузить его критерии повторных заказов в ассоциативный массив PL/SQL. Сделаем это при помощи такого курсора:

```
CURSOR curs_get_criteria ( cp_region NUMBER ) IS
  SELECT *
    FROM repeat_order_criteria
   WHERE region_id = cp_region;
```

Теперь выполним в теле функции простую проверку «последнего ID региона», определяющую, изменилось ли значение, и если да, то выполним массовую загрузку критериев:

```
-- если это новый регион...
IF NVL(v_last_region,0) <> v_order.region_id THEN

  -- установить локальный ID региона и загрузить его критерии
  v_last_region := v_order.region_id;
  OPEN curs_get_criteria(v_order.region_id);
  FETCH curs_get_criteria BULK COLLECT INTO v_criteria;
  CLOSE curs_get_criteria;

END IF;  -- новый регион
```

Здесь проявляется еще одно преимущество табличных функций – возможность раздельного доступа к данным внутри самой функции. Это означает, что доступ к базе данных для получения нарядов может быть сконцентрирован в одном запросе, а для получения критериев – в другом.

Вернемся к нашей функции, псевдокод которой после добавления запроса критериев выглядит так:

```
BEGIN

  -- для каждого наряда...
  LOOP

    FETCH p_curs INTO v_order;
    EXIT WHEN p_curs%NOTFOUND;

    IF первая_запись OR новый_регион THEN
      Загрузить критерии для региона
    END IF;

    IF повторный_наряд THEN
      PIPE ROW();
    END IF;

  END LOOP;  -- для каждого наряда
```

```
  RETURN;
END;
```

Определение потенциальных повторов

Теперь займемся немного более сложной задачей – поиском повторных нарядов. Используем для этого две отдельные операции. Первая операция на основании сравнения типа и даты наряда с первым типом наряда в критерии определит, может ли наряд потенциально иметь повтор. Вторая операция просмотрит остальные наряды по тому же адресу и выберет действительно повторные, сравнив их тип с типом повторного наряда в критерии.

Объясним подробнее на конкретном примере. Рассмотрим такую запись в таблице критериев:

```
START_DAT FIRST_TYPE_ID REPEAT_TYPE_ID
--------- ------------- --------------
19-APR-05           801             87
```

Эта запись говорит о том, что если начиная с 19 апреля 2005 года в течение 30 дней за нарядом типа 801 последует по тому же адресу наряд типа 87, то это будет повторный наряд.

Теперь рассмотрим следующие три наряда.

```
ORDER_NUMBER ORDER_DAT   TYPE_ID LOCATION_ID
------------ --------- ---------- ----------
        1016 19-APR-05        801        343
        1863 20-APR-05         87        343
        2228 21-APR-05         87        343
```

При обработке нашей функцией наряд номер 1016 по адресу 343, имеющий тип 801, сначала будет отмечен, как кандидат на наличие повтора. Все последующие наряды типа 87 по адресу 343 в течение 30 дней будут считаться фактическими повторными нарядами. То есть в качестве действительных повторных нарядов наша функция выберет наряды с номерами 1863 и 2228.

Для нахождения фактических повторных нарядов надо сначала отобрать кандидатов на повтор. Для большей стройности кода выделим логику поиска «потенциальных повторов» в отдельную функцию с именем `load_potential_repeat`. Сначала приведем код, а затем разберемся, как он работает.

```
/*-------------------------------------------------------------------*/
PROCEDURE load_potential_repeat ( p_location_id NUMBER,
                                  p_type_id     NUMBER,
                                  p_date     DATE ) IS
/*-------------------------------------------------------------------*/
  v_hash NUMBER;
BEGIN
```

```
  -- для каждого критерия...
  FOR counter IN 1..v_criteria.LAST LOOP

    -- если тип наряда присутствует в списке критериев
    IF v_criteria(counter).first_type_id = p_type_id THEN

      -- если дата соответствует дате критерия
      IF v_criteria(counter).start_date <= p_date THEN

        -- вычислить хеш для адреса и пары нарядов критерия
        v_hash := DBMS_UTILITY.GET_HASH_VALUE(p_location_id || ':' ||
                          v_criteria(counter).first_type_id || ':' ||
                          v_criteria(counter).repeat_type_id,
                          -32767,65533);

        -- если этого критерия еще нет в списке потенциальных повторов,
        -- то занести его туда
        IF NOT v_potential_repeat.EXISTS(v_hash) THEN
          v_potential_repeat(v_hash).location_id := p_location_id;
          v_potential_repeat(v_hash).first_type_id :=
                              v_criteria(counter).first_type_id;
              v_potential_repeat(v_hash).repeat_type_id :=
                             v_criteria(counter).repeat_type_id;
        END IF;

      END IF;  -- дата соответствует

    END IF;  -- тип наряда присутствует в списке

  END LOOP;  -- для каждого критерия

END load_potential_repeat;
```

Это может показаться немного запутанным, но на самом деле это очень простой алгоритм. Для каждого критерия в данном регионе надо задать такие вопросы:

- Совпадает ли первый тип наряда в критерии с типом обрабатываемого наряда? Если да, то продолжить.
- Предшествует ли дата ввода критерия дате обрабатываемого наряда? Если да, то продолжить.
- Рассчитать хеш-значение на основе адреса наряда и пары типов нарядов, входящих в критерий.
- Присутствует ли в ассоциативном массиве элемент с рассчитанным хеш-значением? Если нет, то продолжить.
- Добавить в ассоциативный массив элемент, содержащий адрес и пару типов нарядов критерия, использовав в качестве индекса полученное хеш-значение.

Если, например, найдены три отдельных потенциальных повтора, то наш ассоциативный массив может выглядеть так:

```
INDEX LOCATION_ID FIRST_TYPE_ID REPEAT_TYPE_ID
----- ----------- ------------- --------------
```

```
-3421          874          1876           202
  -99         1098             2            18
88862           18           100            88
```

В этом случае любые последующие наряды с типом 202 и адресом 874 будут признаны повторными, равно как и наряды типов 18 и 88 по адресам 1098 и 18 соответственно.

Ну а теперь приступим к поиску фактических повторов.

Определение фактических повторов

Найти фактические повторы несложно. Если тип наряда совпадает с типом повтора в какой-либо записи критерия и если имеется соответствующий элемент в ассоциативном массиве потенциальных повторов, то значит, найден фактический повторный наряд. Эту часть также выделим в отдельную функцию.

```
/*---------------------------------------------------------------------*/
FUNCTION order_is_a_repeat ( p_location_id NUMBER,
                             p_type_id     NUMBER,
                             p_date        DATE )
         RETURN NUMBER IS
/*---------------------------------------------------------------------*/
  v_hash NUMBER;
BEGIN

  -- для каждого критерия...
  FOR counter IN 1..v_criteria.LAST LOOP

    -- если тип наряда совпадает с повторным типом в записи критерия
    IF v_criteria(counter).repeat_type_id = p_type_id THEN

      -- вычислить хеш для адреса и пары нарядов критерия
      v_hash := DBMS_UTILITY.GET_HASH_VALUE(p_location_id || ':' ||
                        v_criteria(counter).first_type_id || ':' ||
                        v_criteria(counter).repeat_type_id,
                        -32767,65533);

      -- если есть запись о потенциальном повторе, считаем,
      -- что повтор имеет место
      IF v_potential_repeat.EXISTS(v_hash) THEN
        RETURN(v_hash);
      END IF;

    END IF;  -- типы нарядов совпадают

  END LOOP;  -- для каждого критерия

  RETURN(NULL);

END order_is_a_repeat;
```

Приведем описание алгоритма. Для каждой записи таблицы критериев данного региона:

- Совпадает ли тип обрабатываемого наряда с типом повторного наряда критерия? Если да, продолжить.
- Рассчитать хеш-значение на основе адреса наряда и пары типов нарядов, входящих в критерий.
- Если в ассоциативном массиве потенциальных повторов присутствует элемент с индексом, равным рассчитанному хеш-значению, то найден фактический повторный наряд.
- Вернуть полученное хеш-значение, чтобы соответствующая строка могла быть найдена и сразу отправлена в конвейер.

Окончательная функция

Теперь, когда все составляющие части имеются, взглянем еще раз на псевдокод нашей функции.

```
BEGIN

  -- для каждого наряда...
  LOOP

    FETCH p_curs INTO v_order;
    EXIT WHEN p_curs%NOTFOUND;

    IF первая_запись OR новый_регион THEN
      загрузить критерии для региона
    END IF;

    IF это потенциальный повтор, добавить его в ассоциативный массив.
    IF повторный_наряд then
      PIPE ROW();
    END IF;

  END LOOP;  -- для каждого наряда

  RETURN;
END;
```

Полностью код функции доступен на сайте этой книги в файле `repeat_orders.sql`.

Выполнение функции

Функция выполняется в SQL-операторе `SELECT`, который передает ей другой оператор `SELECT`. Я понимаю, что это может потребовать определенного навыка, но поверьте – оно того стоит. Вот код SQL, вызывающий эту функцию:

```
/* Файл на веб-сайте: repeat_orders.sql */
SQL> SELECT *
  2    FROM TABLE(repeat_order_finder(CURSOR(
  3                 SELECT order_number,
  4                        order_date,
  5                        region_id,
```

```
  6                       type_id,
  7                       location_id
  8                  FROM orders
  9                 WHERE order_date >= SYSDATE - 30
 10              )))
 11  /

 REGION_ID LOCATION_ID FIRST_TYPE_ID REPEAT_TYPE_ID
---------- ----------- ------------- --------------
         1           1             1              2
         2           2             2              3
         3           3             3              4
         4           4             4              5
         4           4             4              5
         5           5             5              6
         6           6             6              7
         7           7             7              8
         8           8             8              9
         9           9             9             10
        10          10            10             11

11 rows selected.
```

Результирующее множество обрабатывается так же, как если бы было получено из таблицы или представления Oracle. Оно может быть ограничено условиями, например «WHERE first_type_id = 3». Еще более ценно здесь то, что результаты некоего замысловатого бизнес-процесса можно получить простым SQL-запросом, позволяющим легко строить отчеты. Вся бизнес-логика, осуществляющая отбор записей, реализуется в базе данных.

Конвейеризация функции позволяет обрабатывать записи в процессе их поступления, сберегая еще больше драгоценного времени.

Функция суммирования

Теперь напишем еще одну табличную функцию для суммирования повторных нарядов по регионам, а затем встроим ее в SQL-код из предыдущего раздела. Новая функция выглядит так:

```
/* Файл на веб-сайте: repeat_orders_summary.sql */
CREATE OR REPLACE FUNCTION summarize_repeat_orders ( p_curs
cursors.repeat_summary_curs )
                RETURN repeat_summary_t
                PIPELINED
                 PARALLEL_ENABLE ( PARTITION p_curs BY RANGE(region_id) ) AS

  v_summary_rec cursors.repeat_summary_rec;
  v_last_region NUMBER;
  v_count       NUMBER := 0;

BEGIN

  -- для каждого повторного наряда
```

```
  LOOP

    -- извлечь повторный наряд
    FETCH p_curs INTO v_summary_rec;
    EXIT WHEN p_curs%NOTFOUND;

    -- если это первая запись, то установить локальный ID региона
    IF p_curs%ROWCOUNT = 1 THEN
      v_last_region := v_summary_rec.region_id;
    END IF;

    -- если это новый регион, передать в конвейер счетчик регионов
    -- и переустановить локальные переменные
    IF v_summary_rec.region_id <> v_last_region THEN
      PIPE ROW(repeat_summary_o(v_last_region,v_count));
      v_last_region := v_summary_rec.region_id;
      v_count := 0;
    END IF;

    v_count := v_count + 1;

  END LOOP;  -- для каждого повторного наряда

  -- не забыть про последнюю запись
  IF v_count > 0 THEN
    PIPE ROW(repeat_summary_o(v_last_region,v_count));
  END IF;

  RETURN;

END;
```

Алгоритм состоит из простого цикла по выбранным повторным нарядам, в котором они суммируются по регионам. Как только в наряде меняется значение идентификатора региона, результат отправляется в конвейер.

Функция суммирования вызывается из оператора SELECT, как показано ниже.

```
SQL> SELECT *
  2    FROM TABLE(summarize_repeat_orders(CURSOR(
  3         SELECT *
  4          FROM TABLE(repeat_order_finder(CURSOR(
  5               SELECT order_number,
  6                      order_date,
  7                      region_id,
  8                      type_id,
  9                      location_id
 10                 FROM orders
 11                WHERE order_date >= SYSDATE - 30
 12                    )))
 13              )));

 REGION_ID REPEAT_COUNT
---------- ------------
```

```
         1          1
         2          1
         3          1
         4          2
         5          1
         6          1
         7          1
         8          1
         9          1
        10          1

10 rows selected.
```

Представленный здесь способ (использование нескольких табличных функций) иллюстрирует упоминавшуюся выше технологию вложения (или организации цепочки) табличных функций. Работа распределяется между несколькими функциями, передающими результаты по цепочке, запись за записью, пока не будет собрано окончательное результирующее множество. С учетом параллельного режима работы этих функций выгода от использования табличных функций становится очевидной.

Примеры табличных функций

В этом разделе приведен ряд дополнительных примеров, демонстрирующих удобство использования табличных функций в таких задачах, как дополнительная трассировка, установка временных ограничений и периодическое обновление данных. Во всех примерах используется такое полезное свойство табличных функций, как возможность разместить код в операторе SELECT.

Трассировка

Большинство средств трассировки PL/SQL (SQL Trace, DBMS_TRACE и т. п.) устроены так, что после выполнения операции вам надо где-то искать результаты трассировки. Даже пакет Oracle DBMS_OUTPUT (простейший отладочный инструмент) требует отдельного места для вывода при использовании средств разработки типа Toad или PL/SQL Developer.

Табличные функции позволяют включать отладочную информацию в результаты запроса. В сочетании с автономными транзакциями они помогают трассировать даже операции DML. Рассмотрим следующую функцию:

```
/* Файл на веб-сайте: tracer.sql */
CREATE OR REPLACE FUNCTION tracer
                  RETURN debug_t AS
  PRAGMA AUTONOMOUS_TRANSACTION;
  v_debug debug_t := debug_t();
BEGIN
  v_trace.EXTEND;
```

```
  v_trace(v_debug.LAST) := 'Started Insert At ' ||
                           TO_CHAR(SYSDATE,'HH24:MI:SS');
  INSERT INTO a_table VALUES(1);
  COMMIT;
  v_trace.EXTEND;
  v_trace(v_debug.LAST) := 'Completed Insert At ' ||
                           TO_CHAR(SYSDATE,'HH24:MI:SS');
  RETURN(v_trace);
END;
```

Без предложения AUTONOMOUS TRANSACTION мы получили бы при выполнении запроса ошибку «ORA-14551: cannot perform a DML operation inside a query» (невозможно выполнить операцию DML внутри запроса). Если это предложение присутствует, функцию можно выполнить из оператора SELECT.

```
SQL> SELECT *
  2    FROM a_table;

no rows selected

SQL> SELECT *
  2    FROM TABLE(tracer);

COLUMN_VALUE
----------------------------
Started Insert At 22:04:28
Completed Insert At 22:04:28

SQL> SELECT *
  2    FROM a_table;

      COL1
----------
         1
```

Установка временных ограничений

Табличные функции могут быть очень полезны при установке ограничения на время выборки записей запросом. Это удобно, если вы хотите протестировать свое приложение на некотором подмножестве записей и не хотите ждать, пока будет выбран полный список. Следующая функция передает полученные запросом данные в конвейер в течение заданного параметром количества секунд. Как только заданное время истекает, возвращается значение –1 и функция завершается.

```
/* Файл на веб-сайте: time_limit.sql */
CREATE OR REPLACE FUNCTION get_a_table
          ( p_limit NUMBER )
          RETURN rowset_t
          PIPELINED AS
  CURSOR curs_get_a IS
  SELECT *
```

```
    FROM a_table;
  v_start DATE;
BEGIN
  v_start := SYSDATE;
  FOR v_a_rec IN curs_get_a LOOP
    PIPE ROW(rowset_o(v_a_rec.col1));
    IF SYSDATE - v_start >= ( p_limit * 0.000011574 ) THEN
      PIPE ROW(rowset_o(-1));
      EXIT;
    END IF;
  END LOOP;
END;
```

Вот пример выборки из таблицы, содержащей 1000 записей:

```
SQL> SELECT *
  2    FROM TABLE(get_a_table(1));

      COL1
----------
       661
       662
       663
       664
        -1

5 rows selected.
```

Если запрос требует для выполнения больше времени, чем `p_limit` секунд, то функция выйдет за пределы лимита времени.

Использование вложенных курсоров

С помощью табличных функций, кроме всего прочего, можно, используя знания о приложении, помочь выполнению запросов. Классический пример запроса с несколькими условиями `OR EXISTS` показан в следующем примере.

```
SELECT *
  FROM main_table mt
  WHERE col1 = 1
    AND ( EXISTS ( SELECT 1
                     FROM or_table_one
                    WHERE col11 = mt.col1 )
       OR EXISTS ( SELECT 1
                     FROM or_table_two
                    WHERE col21 = mt.col1 )
       OR EXISTS ( SELECT 1
                     FROM or_table_three
                    WHERE col31 = mt.col1 ) );
```

Запрос вернет запись, если соответствующая запись будет найдена в любой из трех других таблиц. Полученный в режиме `AUTOTRACE` план выполнения показывает, что оптимизатор Oracle использует просмотр

всех использованных в запросе таблиц для получения единственной результирующей записи.

```
Execution Plan
----------------------------------------------------------
     0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1 Card=1 Bytes=4)
   1    0   FILTER
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'MAIN_TABLE' (TABLE) (Cost=1
               Card=1 Bytes=4)
   3    2      INDEX (UNIQUE SCAN) OF 'SYS_C003477' (INDEX (UNIQUE)) (Cost=0
               Card=1)
   4    1     INDEX (UNIQUE SCAN) OF 'SYS_C003479' (INDEX (UNIQUE)) (Cost=0
               Card=1 Bytes=2)
   5    1     INDEX (UNIQUE SCAN) OF 'SYS_C003481' (INDEX (UNIQUE)) (Cost=1
               Card=1 Bytes=3)
   6    1     INDEX (UNIQUE SCAN) OF 'SYS_C003483' (INDEX (UNIQUE)) (Cost=1
               Card=1 Bytes=3)

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
         13  consistent gets
          0  physical reads
          0  redo size
        446  bytes sent via SQL*Net to client
        511  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Секция статистики показывает объем работы, проделанной Oracle для получения результирующего множества, включая 13 операций согласованного чтения.

Но как быть, если нам известно, что для таблицы OR_TABLE_ONE условие выполняется с вероятностью 90%, а для остальных таблиц вероятность составляет 10%? Надо выполнить поиск в таблице OR_TABLE_ONE и, только если там запись не найдена, приступать к поиску в других таблицах. Одно из решений заключается в использовании в табличной функции вложенных курсоров, с тем чтобы вся операция по-прежнему выполнялась в виде запроса.

```
CREATE OR REPLACE FUNCTION nested
                  RETURN number_t AS

  -- получить запись из главной таблицы
  CURSOR curs_get_mt IS
  SELECT mt.col1,
         CURSOR ( SELECT 1
                    FROM or_table_one
                   WHERE col11 = mt.col1 ),
         CURSOR ( SELECT 1
```

```
                  FROM or_table_two
                 WHERE col21 = mt.col1 ),
       CURSOR ( SELECT 1
                  FROM or_table_three
                 WHERE col31 = mt.col1 )
  FROM main_table mt
 WHERE col1 = 1;

 v_col1       NUMBER;
 cursor_one   SYS_REFCURSOR;
 cursor_two   SYS_REFCURSOR;
 cursor_three SYS_REFCURSOR;
 v_dummy      NUMBER;

 v_ret_val number_t := number_t();

BEGIN

 OPEN curs_get_mt;
 FETCH curs_get_mt INTO v_col1,
                        cursor_one,
                        cursor_two,
                        cursor_three;

 IF curs_get_mt%FOUND THEN
   -- поиск в первой таблице OR
   FETCH cursor_one INTO v_dummy;
   IF cursor_one%FOUND THEN
     v_ret_val.EXTEND;
     v_ret_val(v_ret_val.LAST) := v_col1;
   ELSE
     -- поиск во второй таблице OR
     FETCH cursor_two INTO v_dummy;
     IF cursor_two%FOUND THEN
       v_ret_val.EXTEND;
       v_ret_val(v_ret_val.LAST) := v_col1;
     ELSE
       - поиск в третьей таблице OR
       FETCH cursor_three INTO v_dummy;
       IF cursor_two%FOUND THEN
         v_ret_val.EXTEND;
         v_ret_val(v_ret_val.LAST) := v_col1;
       END IF;
     END IF;
   END IF;
 END IF;
 IF cursor_one%ISOPEN THEN
   CLOSE cursor_one;
 END IF;
 IF cursor_two%ISOPEN THEN
   CLOSE cursor_two;
 END IF;
 IF cursor_three%ISOPEN THEN
   CLOSE cursor_three;
```

```
  END IF;
  CLOSE curs_get_mt;
  RETURN(v_ret_val);
END;
```

План выполнения при включенном AUTOTRACE для этой функции выглядит так:

```
SQL> SELECT *
  2    FROM TABLE(nested);

COLUMN_VALUE
------------
           1

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=25 Card=8168 Bytes=16336)
   1    0   COLLECTION ITERATOR (PICKLER FETCH) OF 'NESTED'

Statistics
----------------------------------------------------------
         13  recursive calls
          0  db block gets
          3  consistent gets
          0  physical reads
          0  redo size
        397  bytes sent via SQL*Net to client
        511  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Рассчитанная оптимизатором стоимость выполнения такой функции намного выше, чем у запроса, – в 25 раз. Но базе данных приходится выполнять гораздо меньше работы, всего 3 операции согласованного чтения вместо 13.

Небольшое предостережение: используйте такой способ, только если вы обнаружили узкое место и только после тщательного тестирования. Выигрыш в производительности может не стоить написания, отладки и сопровождения дополнительного кода.

Советы по работе с табличными функциями

В завершение дадим несколько советов, которые помогут вам полнее использовать преимущества табличных функций.

Критика SYS_REFCURSOR

Функция Oracle SYS_REFCURSOR позволяет быстро объявить слабо типизированный параметр REF CURSOR, которому можно сопоставить практи-

чески любой курсор. Используя SYS_REFCURSOR для объявления типа данных параметра, в функцию можно передавать любой оператор SELECT при условии, что над курсором в теле функции не выполняется никаких действий. Например, эта функция может принять любой оператор SELECT.

```
CREATE OR REPLACE FUNCTION wide_open ( p_curs SYS_REFCURSOR )
                  RETURN number_t IS
  v_ret_val number_t := number_t();
BEGIN
  v_ret_val.EXTEND;
  v_ret_val(v_ret_val.LAST) := 99;
  RETURN v_ret_val;
END;
```

С любым корректным оператором SELECT она будет работать.

```
SQL> SELECT *
  2    FROM TABLE(wide_open(CURSOR(SELECT NULL
  3                                FROM DUAL)));

COLUMN_VALUE
------------
          99

SQL> SELECT *
  2    FROM TABLE(wide_open(CURSOR(SELECT *
  3                                FROM orders)));

COLUMN_VALUE
------------
          99
```

Ограничения появляются, когда внутри функции, что весьма вероятно, надо извлекать записи. Для сохранения записей потребуются локальные переменные.

```
CREATE OR REPLACE FUNCTION wide_open ( p_curs SYS_REFCURSOR )
                  RETURN number_t IS
  v_ret_val   number_t := number_t();
  v_order_rec orders%ROWTYPE;
BEGIN
  FETCH p_curs INTO v_order_rec;
  v_ret_val.EXTEND;
  v_ret_val(v_ret_val.LAST) := 99;
  RETURN v_ret_val;
END;
```

Теперь в функцию могут быть переданы только те операторы SELECT, которые выбирают все столбцы таблицы ORDERS. Любые другие вызовут ошибку ORA-01007.

```
SQL> SELECT *
  2    FROM TABLE(wide_open(CURSOR(SELECT NULL
  3                                FROM DUAL)));
```

```
  FROM TABLE(wide_open(CURSOR(SELECT NULL
             *
ERROR at line 2:
ORA-01007: variable not in select list
ORA-06512: at "SCOTT.WIDE_OPEN", line 6

SQL> SELECT *
  2    FROM TABLE(wide_open(CURSOR(SELECT *
  3                                FROM orders)));

COLUMN_VALUE
------------
          99
```

Таким образом, в функцию можно передать любой оператор SELECT, но если в нем запрашивается таблица, отличная от ORDERS, функция завершится с ошибкой. Гибкость SYS_REFCURSOR и слабо типизированных курсоров REF CURSOR в общем случае оказывается сомнительной.

В силу сказанного, мы рекомендуем полностью избавиться от иллюзии гибкости и использовать курсоры REF CURSOR со строгим контролем типа, объявленные в централизованном пакете, как показано ниже:

```
CREATE OR REPLACE PACKAGE cursors AS
  TYPE order_curs IS REF CURSOR RETURN orders%ROWTYPE;
END;
```

и в дальнейшем использовать их в табличных функциях.

```
CREATE OR REPLACE FUNCTION wide_open ( p_curs cursors.order_curs )
                  RETURN number_t IS
  v_ret_val   number_t := number_t();
  v_order_rec p_curs%ROWTYPE;
BEGIN
  FETCH p_curs INTO v_order_rec;
  v_ret_val.EXTEND;
  v_ret_val(v_ret_val.LAST) := 99;
  RETURN v_ret_val;
END;
```

Дополнительное преимущество такому подходу дает привязка типа данных локальной переменной, в которую извлекаются записи, непосредственно к типу строго типизированного курсора. Благодаря этому экономится время выполнения, так как Oracle не придется заниматься выяснением структуры возвращаемых данных. Помимо этого, появляется возможность координировать операторы SELECT и параметры курсоров с помощью централизованного пакета – в том случае, если возникнет желание что-либо изменить. Не придется менять каждую функцию для сохранения согласованности.

REF CURSOR и вложение

Табличные функции возвращают коллекции, поэтому нет простого способа объявить строго типизированный REF CURSOR, чтобы использо-

вать его при вложении. Поэтому приходится объявлять тип RECORD с той же структурой, что и у коллекции, а затем связывать с ней REF CURSOR таким способом:

```
CREATE OR REPLACE PACKAGE cursors
  TYPE v_number_rec IS RECORD ( number_col NUMBER );
 TYPE number_curs IS REF CURSOR RETURN v_number_rec;
END;
```

Теперь REF CURSOR со строгим контролем типа можно использовать во вложенной табличной функции.

```
CREATE OR REPLACE FUNCTION nested_number ( p_curs cursors.number_curs )...
```

Использование условий

Не забывайте о производительности, когда применяете условия к функциям, особенно когда решаете, передать ли значения параметрами в функцию, чтобы они там использовались для формирования результирующего множества, или применить их к возвращаемому результирующему множеству. Вот два примера, поясняющие эту мысль. В первом из них условие (col1 = 'A') применяется к возвращаемому множеству записей после того, как оно было сформировано функцией.

```
SELECT *
  FROM TABLE(a_function)
 WHERE col1 = 'A';
```

Во втором примере значение передается прямо в функцию и там используется для формирования результирующего множества.

```
SELECT *
  FROM TABLE(a_function('A');
```

Оцените сложность алгоритма и объем данных, чтобы решить, какой из подходов предпочтительнее в вашем случае.

Стандартизация имен объектов и коллекций

Разработав несколько приложений с использованием табличных функций, я с тревогой заметил, как растет количество повторяющихся типов объектов и коллекций. Например, я создал два таких объекта:

```
SQL> DESC experiment_results_o
 Name                                      Null?    Type
 ----------------------------------------- -------- ------
 SAMPLE_AMT                                         NUMBER

SQL> DESC research_tallies_o
 Name                                      Null?    Type
 ----------------------------------------- -------- ------
 TALLY_TOTAL                                        NUMBER
```

Затем создал на их основе коллекции с соответствующими именами, заменив суффикс «_o» на «_t». Это самый простой пример той неразбе-

рихи, которую я создал, сосредоточившись на частностях, а не базе данных в целом. Потом я вернулся к этому месту и заменил эти два объекта одним:

```
SQL> DESC number_o
 Name                                      Null?    Type
 ----------------------------------------- -------- ------
 COL1                                                NUMBER
```

У меня есть аналогичные базовые объекты и для других типов данных, включая поля VARCHAR2 нескольких стандартных размеров.

Еще один стандарт, которого я придерживаюсь, – это использование суффикса «_o» в именах объектов и суффикса «_t» в именах коллекций (или таблиц). Это позволяет мне быстро различать их типы.

Остерегайтесь необработанных исключений

Обработка исключений в функциях, помещенных в оператор SELECT, требует особого внимания. Здесь это несколько сложнее, чем простая передача ошибки в вызывающую программу. Например, что делать в ситуации, когда в приведенном примере функция порождает исключение NO DATA FOUND?

```
CREATE OR REPLACE FUNCTION unhandled
                  RETURN number_t AS
  v_ret_val number_t := number_t();
  v_dummy   NUMBER;
BEGIN
  SELECT 1
    INTO v_dummy
    FROM DUAL
   WHERE 1 = 2;
  v_ret_val.EXTEND;
  v_ret_val(v_ret_val.LAST) := 1;
  RETURN(v_ret_val);
END;
```

Должен ли оператор SELECT вернуть исключение в таком виде?

```
SQL> SELECT *
  2   FROM TABLE(unhandled);

COLUMN_VALUE
------------------------
ORA-01403: no data found
```

В этом случае Oracle придется поддерживать две структуры выходных данных: одну для успешного выполнения, а вторую, состоящую из одного столбца VARCHAR2, для возможного сообщения об ошибке. Такой подход возможен, но он стал бы источником проблем для вложенных табличных функций, так как им тоже пришлось бы обрабатывать выходные данные, имеющие две различные структуры. Это было бы слишком сложно.

Может быть, SELECT просто завершится с ошибкой?

```
SQL> SELECT *
  2    FROM TABLE(unhandled);

ORA-01403: no data found
```

Это лучше, чем возвращать сообщение об ошибке, но иногда может сбивать с толку.

Решение, применяемое Oracle, заключается в том, что функция, в которой возникла ошибка, просто не возвращает строк.

```
SQL> SELECT *
  2    FROM TABLE(unhandled);

no rows selected
```

Надо очень тщательно обрабатывать все возможные исключения, иначе ваша табличная функция может не сообщить о возникшей ошибке.

Передача объектов вместо курсоров

Не все знают, что табличные функции могут в качестве параметров принимать не только курсоры, но и коллекции. Вот простой пример:

```
CREATE OR REPLACE FUNCTION give_me_a_collection ( p_col number_t )
                  RETURN number_t IS
  v_ret_val number_t := number_t();
BEGIN
  v_ret_val.EXTEND(p_col.COUNT);
  FOR counter IN v_ret_val.FIRST..v_ret_val.LAST LOOP
    v_ret_val(counter) := p_col(counter);
  END LOOP;
  RETURN(v_ret_val);
END;
```

И вот как он выполняется в операторе SELECT:

```
SQL> SELECT *
  2    FROM TABLE(give_me_a_collection(number_t(1,2,3)));

COLUMN_VALUE
------------
           1
           2
           3
```

Отсутствие режима Read Committed

Несмотря на то что табличные функции выполняются внутри оператора SELECT, они в ходе выполнения не могут воспользоваться преимуществами используемой в Oracle архитектуры «read-committed» (чтение только зафиксированных данных). Любым запросам, выполняемым внутри табличной функции, режим «read-committed» доступен, но сами табличные функции в этом отношении работают так же, как и любые другие функции. Рассмотрим такую табличную функцию:

```
CREATE OR REPLACE FUNCTION not_committed
                  RETURN number_t IS
  v_ret_val NUMBER_T := NUMBER_T();
  v_count   NUMBER;
BEGIN
  SELECT COUNT(*)
    INTO v_count
    FROM orders;
  v_ret_val.EXTEND;
  v_ret_val(v_ret_val.LAST) := v_count;
  DBMS_LOCK.SLEEP(10);
  SELECT COUNT(*)
    INTO v_count
    FROM orders;
  v_ret_val.EXTEND;
  v_ret_val(v_ret_val.LAST) := v_count;
  RETURN(v_ret_val);
END;
```

Она запрашивает количество строк в таблице ORDERS, ждет 10 секунд и повторяет запрос, возвращая результирующее множество из этих двух значений. Если вы выполните эту функцию в одном сеансе, затем удалите (и зафиксируете удаление) 5 заказов в другом сеансе (предположим, это происходит в рассматриваемом 10-секундном интервале), то получите следующие результаты:

```
SQL> SELECT *
  2    FROM TABLE(not_committed);

COLUMN_VALUE
------------
       10000
        9995
```

Имейте в виду эту особенность, когда будете выбирать между табличными функциями и запросами. Если вы считаете, что доступ в режиме «read-committed» принципиально важен, то, возможно, от табличных функций лучше отказаться.

Заключение

Когда вы используете табличную функцию, программа становится таблицей. Что я хочу этим сказать? Фактически программа (ваша табличная функция) выступает в роли источника данных, к которому адресуются запросы, – в точности, как к таблице. Табличные функции отлично подходят для случаев, когда сложная логика должна выполняться непосредственно в операторе SELECT, что часто бывает при взаимодействии Oracle со сторонними приложениями. Объединение этой возможности с мощью параллельного выполнения табличных функций позволяет получить прекрасный инструмент для администраторов баз данных.

4

Шифрование и хеширование данных

Говоря доступным языком, *шифрование* – это сокрытие содержимого, изменение данных таким способом, что знание о том, как вернуть данные в первоначальный вид, доступно только их создателю. В этой главе мы обсудим поддержку шифрования в Oracle, останавливаясь прежде всего на концепциях и возможностях, наиболее интересных администраторам баз данных. Основное внимание будет уделено использованию встроенных пакетов Oracle: `DBMS_CRYPTO` (доступен начиная с версии Oracle 10*g* Release 1) и `DBMS_OBFUSCATION_TOOLKIT` (используется преимущественно с более ранними версиями). Остановимся также на защите данных на диске, не рассматривая защиту данных в процессе передачи между клиентом и сервером и защиту данных в процессе аутентификации (две последние задачи требуют наличия опции Advanced Security Option (ASO), поставляемой за отдельную плату). Исключением является только передача паролей, которые шифруются всегда, независимо от наличия ASO.

В этой главе вы научитесь создавать базовую систему шифрования, защищающую конфиденциальные данные от неавторизованных пользователей. Вы узнаете, как построить систему управления ключами шифрования, обеспечивающую одновременно сохранность ключей и прозрачность доступа к данным для пользователей приложений. Вы также познакомитесь с криптографическим хешированием и научитесь использовать код аутентификации сообщения MAC (Message Authentication Code). Будет описан режим прозрачного шифрования данных TDE (Transparent Data Encryption), появившийся в Oracle 10*g* Release 2 и позволяющий с наименьшими усилиями шифровать важные данные и удовлетворяющий требованиям многочисленных нормативных документов.

В соответствии с рекомендациями Oracle, если вы используете Oracle 10*g*, вам следует перейти к использованию пакета `DBMS_CRYPTO`, отка-

завшись от DBMS_OBFUSCATION_TOOLKIT. Однако, поскольку версия Oracle9*i* Database все еще широко используется, мы сначала изучим пакет DBMS_OBFUSCATION_TOOLKIT, а затем рассмотрим возможности версии Oracle 10*g*. Даже если вы используете новую версию, вам имеет смысл прочитать этот раздел, чтобы убедиться в том, что вы хорошо ориентируетесь в концепциях шифрования.

Пакет DBMS_CRYPTO имеет ряд преимуществ перед DBMS_OBFUSCATION_TOOLKIT:

- Больший выбор алгоритмов шифрования, в частности, поддержка последнего стандарта AES (Advanced Encryption Standard).
- Возможность поточного шифрования, то есть организации потока предназначенных для шифрования данных.
- Поддержка алгоритма SHA-1 (Secure Hash Algorithm 1).
- Способность создания кода MAC.
- Шифрование больших объектов (LOB) в их *собственном* формате.

Все эти возможности будут рассмотрены в данной главе. В приложении А вы найдете краткий справочник по процедурам и функциям пакетов DBMS_CRYPTO и DBMS_OBFUSCATION_TOOLKIT.

В этой книге не рассматриваются подробности криптографических алгоритмов, теория компьютерного шифрования и искусство его применения – эта область требует гораздо более глубокого изучения, чем мы можем здесь себе позволить. Мы стремимся лишь к тому, чтобы читатели могли приступить к созданию защищенной системы на основе встроенных инструментов Oracle, а не изобретали велосипед, занимаясь реализацией уже существующих алгоритмов. Существует множество прекрасных книг, из которых вы можете почерпнуть дополнительные сведения по криптоанализу, математическим основам шифрования и смежным вопросам.

Введение в шифрование

Давайте представим, что вы каждый день уносите с работы домой свой ноутбук, а на следующее утро приносите его обратно в офис и пристегиваете к своему столу кабелем с кодовым замком. Вы ведь понимаете, как важно помнить кодовую комбинацию? Если вдруг вы ее забудете, ваш ноутбук будет прикован к столу, пока вы не перережете кабель. Может быть, вы легко запоминаете цифры, но я лично – нет. Я с трудом запоминаю даже свой телефонный номер, не говоря уже об окружающих меня многочисленных секретных номерах: номере социального страхования, PIN-коде банковского счета, пароле голосовой почты и (увы!) годовщинах. Чтобы справиться с этой проблемой, я нашел гениальный способ запоминания кодовой комбинации: я наклеил этикетку с кодом прямо на замок!

Теперь у вас, наверное, возник вопрос, рискнете ли вы доверить мне какой-нибудь секрет!

Мой мозг, как и у всего остального человечества, включает в себя долговременную память (диск) и оперативное запоминающее устройство (ОЗУ), и, похоже, числа обычно записываются в ОЗУ. Некоторое время числа используются, а затем, чтобы освободить место для новых, помечаются как устаревшие (совсем как в области SGA экземпляра Oracle) и забываются. В компьютерах такое поведение является штатным и закладывается при проектировании. СУБД предназначаются для хранения информации и предоставления ее пользователям по запросу. Исторически считается, что требующие доступа пользователи уже прошли процедуру аутентификации, подтвердившую, что они действительно те, за кого себя выдают. При этом предполагается, что само по себе хранилище конфиденциальной информации не является брешью в системе безопасности.

Возможно, когда-нибудь так и будет, но только не сейчас, когда взломщики, кажется, уже повсюду: возможно, они просто любопытны, а может быть, намереваются продать сведения о состоянии ваших счетов конкурентам или отомстить вам за что-то, разрушив систему. Атака может произойти снаружи, через Интернет, или изнутри вашей организации. (Исследования и в самом деле показывают, что большинство взломов происходят изнутри.) Очевидно, при столь многочисленных угрозах безопасности чувствительные данные должны быть защищены от неавторизованного доступа. Какие же возможности предоставляет Oracle для такой защиты?

Вернемся к моему кодовому замку – его комбинация 3451. Не будучи законченным идиотом, я не стал записывать на своем замке это число. Вместо этого я воспользовался комбинацией, которую помню *всегда* – 6754, и с ее помощью изменил комбинацию, сложив соответствующие цифры:

```
3 + 6 = 9
4 + 7 = 11
5 + 5 = 10
1 + 4 = 5
```

В результате получились числа 9, 11, 10 и 5. В своей схеме я использую только однозначные числа, для этого я сбрасываю в двузначных числах цифру десятков, тогда 10 превращается в 0, 11 превращается в 1 и т. д. При помощи своего секретного ключа 6754 я превратил число 3451 в 9105. *Именно это* число я написал на кодовом замке, а вовсе не исходную комбинацию. Если я забуду ее, то прочитаю написанное на замке число, с помощью своего магического числа 6754 выполню действия, противоположные сделанным ранее, и получу число 3451, которое откроет замок. Число 9105 открыто для всеобщего обозрения, но похититель не сможет открыть замок, пока не узнает еще и ключ, 6754.

Таким образом, я *зашифровал* число, представляющее мою кодовую комбинацию. Число 6754 использовано в процессе шифрования в качестве *ключа*. Этот тип шифрования известен как *симметричное шиф-*

рование, так как для зашифровывания и расшифровывания используется один и тот же ключ. (В противоположность этому при асимметричном шифровании, описанном ниже в этой главе, применяются два ключа: открытый и секретный.) Описанный мной способ шифрования кодовой комбинации представляет собой простейшую реализацию *алгоритма шифрования*.

Компоненты шифрования

Давайте подытожим, что нам на данный момент известно. Система шифрования, как показано на рис. 4.1, включает в себя несколько базовых компонентов:

- Алгоритм
- Ключ
- Тип шифрования (в данном случае симметричное, т. к. для зашифровывания и расшифровывания используется один и тот же ключ)

Предположим, что злоумышленник, решивший украсть мой ноутбук, пытается открыть замок. Что ему нужно для успешной кражи? Во-первых, ему надо знать алгоритм; допустим, он его знает благодаря тому, что я хвастался коллегам своей сообразительностью, или он читал эту книгу, или этот алгоритм широко известен. Во-вторых, ему надо знать ключ. А это то, что я могу защитить. Даже если вору известен алгоритм, я могу надежно спрятать ключ. Но в силу того, что ключ состоит всего лишь из 4 цифр, вору потребуется не более 10^4 (10 000) попыток для его угадывания. И поскольку вероятности угадывания в каждой из попыток равны, то теоретически вору в среднем надо перебрать 5000 комбинаций. Возможно ли это? В нашем случае вору придется *вручную* поворачивать колесики замка 5000 раз. Это непросто, но теоретически возможно. Теперь я уже не чувствую себя в безопасности.

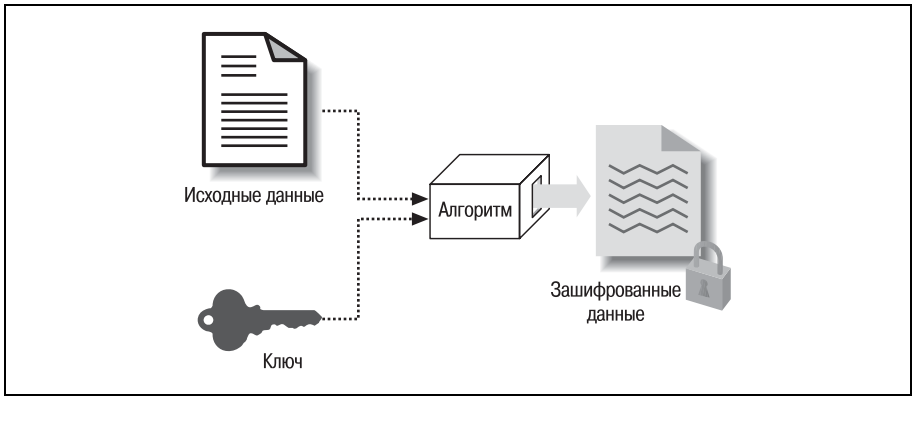


Рис. 4.1. *Компоненты симметричного шифрования*

Какими способами я мог бы защитить кодовую комбинацию своего замка?

- Я могу засекретить алгоритм.
- Я могу сделать ключ трудно угадываемым.
- Я могу применить оба этих метода.

Первый вариант не подходит, если я использую общеизвестный алгоритм. Я могу разработать свой собственный, но потраченные время и силы могут не окупиться. Алгоритм позже может быть раскрыт, а его замена – очень сложная задача. Эти доводы применимы и к третьему варианту, так что практическую ценность представляет только второй.

Влияние длины ключа

Моя кодовая комбинация – это представленные в числовом виде секретные данные. Если злоумышленник решит взломать зашифрованный код, то выполнение 10 000 итераций для его угадывания – тривиальная задача; код будет взломан менее чем за секунду. Что если я использую вместо цифрового ключа алфавитно-цифровой? Это даст 36 возможных значений для каждого символа ключа, так что взломщику придется перебрать не более 36^4 или 1 679 616 комбинаций; это сложнее, чем 10 000, но все еще возможно. Ключ должен быть усилен или сделан более «стойким» за счет увеличения его длины. В табл. 4.1 показано, как возрастает количество попыток, необходимое для угадывания, с увеличением длины ключа. Так что секрет повышения стойкости ключа заключается в увеличении его длины.

Таблица 4.1. Длина алфавитно-цифрового ключа и максимальное число попыток, необходимых для его угадывания

Длина ключа	Максимальное число попыток для угадывания
4	1 679 616
5	60 466 176
6	2 176 782 336
7	78 364 164 096
8	2 821 109 907 456
9	101 559 956 668 416
10	3 656 158 440 062 976

Не забывайте, что компьютеры оперируют битами и байтами (т. е. двоичными числами), а не алфавитно-цифровыми символами. В отдельном разряде ключа может находиться 0 или 1, поэтому 10-разрядный ключ может быть найден за 2^{10} или 1024 попытки, что не представляет никаких трудностей. На практике это означает, что ключ должен быть намного длиннее. Длина ключа выражается в битах, так что ключ из 64 цифр называется 64-битовым. В табл. 4.2 показана зависимость меж-

ду длиной двоичного ключа и максимальным количеством попыток, необходимым для его угадывания.

Таблица 4.2. Длина двоичного ключа и максимальное число попыток, необходимых для его угадывания

Длина ключа	Максимальное число попыток для угадывания
56	72 057 594 037 927 936
57	144 115 188 075 855 872
58	288 230 376 151 711 744
59	576 460 752 303 423 488
60	1 152 921 504 606 846 976
61	2 305 843 009 213 693 952
62	4 611 686 018 427 387 904
63	9 223 372 036 854 775 808
64	18 446 744 073 709 551 616
65	36 893 488 147 419 103 232

Чем длиннее ключ, тем труднее взломать шифр. Но длинные ключи требуют больше времени для шифрования, так как процессору приходится выполнять больше работы. При проектировании инфраструктуры шифрования вам, возможно, придется искать компромисс между длиной ключа и степенью безопасности.

Сравнение симметричного и асимметричного шифрования

В приведенном выше примере для зашифровывания и расшифровывания использовался один и тот же ключ. Как уже говорилось, шифрование такого типа называется *симметричным шифрованием*. При таком шифровании возникает одна неизбежная трудность: из-за того, что данные расшифровываются тем же ключом, этот ключ должен быть известен получателю. Такой ключ, обычно называемый *секретным ключом*, получатель должен либо узнать до получения зашифрованных данных (то есть должно иметь место «соглашение об обмене знаниями»), либо ключ должен пересылаться вместе с данными. Для хранимых данных (на диске) такой ключ должен храниться как часть базы данных, чтобы приложения имели возможность расшифровать их. Риски, возникающие в такой ситуации, очевидны. Пересылаемый ключ может быть перехвачен взломщиком, а ключ, хранящийся в базе данных, может быть украден.

Для решения данной проблемы часто используется шифрование другого типа, при котором ключ, применяемый для шифрования, отличается от ключа, используемого при расшифровывании. Из-за различия

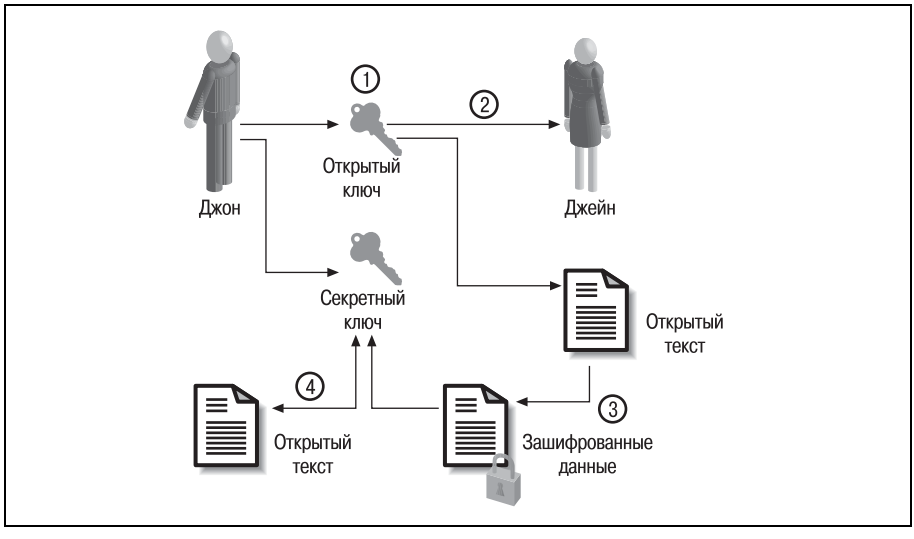


Рис. 4.2. *Базовая схема асимметричного шифрования*

ключей такое шифрование называется *асимметричным*. Вследствие использования двух ключей – секретного и открытого – оно также называется *шифрованием открытым ключом*. Этот *открытый ключ*, необходимый при шифровании, известен отправителю и фактически может свободно распространяться. Другой, *секретный ключ*, нужен только для расшифровывания данных, зашифрованных открытым ключом, и должен храниться в тайне.

Посмотрим, как шифрование открытым ключом может работать в реальной жизни. На рис. 4.2 показано, что Джон ожидает сообщение от Джейн. Процесс шифрования состоит из нескольких шагов:

1. Джон создает два ключа: открытый и секретный.
2. Он отправляет открытый ключ Джейн.
3. Джейн зашифровывает исходное сообщение (называемое *открытым текстом*) с помощью открытого ключа и посылает зашифрованное сообщение Джону.
4. Джон расшифровывает его при помощи сгенерированного им ранее секретного ключа.

Обратите внимание, что здесь стороны не обмениваются расшифровывающим ключом. Открытый ключ посылается отправителю, но поскольку как этот ключ не годится для расшифровывания данных, его возможный перехват не создает никакой угрозы.

Следует, однако, опасаться попыток спуфинга и фишинга, способных нарушить безопасность данного способа шифрования. Сценарий может быть таким:

1. Джон создает пару из открытого и секретного ключей и передает открытый ключ Джейн.
2. Взломщик прослушивает канал связи и перехватывает открытый ключ Джона. Иногда даже этого не требуется, так как Джон может намеренно сделать открытый ключ общедоступным.
3. Взломщик с помощью своих программ создает *другую* пару из открытого и секретного ключей (используя имя Джона, чтобы имитировать принадлежность открытого ключа ему).
4. Взломщик отправляет сгенерированный им новый открытый ключ вместо исходного, созданного Джоном. Джейн ничего не знает о подмене и считает, что это подлинный ключ Джона.
5. Джейн шифрует этим открытым ключом сообщение и отправляет его Джону.
6. Однако взломщик по-прежнему прослушивает канал и перехватывает это сообщение. У него есть секретный ключ, соответствующий открытому, поэтому он может расшифровать сообщение. Через мгновение преимущества секретности сведены на нет.
7. Правда, остается небольшая проблема. Когда Джон в конце концов получает зашифрованное сообщение и пытается его расшифровать, его ждет разочарование: имеющийся у него секретный ключ оказывается неподходящим. Это вызовет подозрение. Во избежание этого взломщику надо лишь повторно зашифровать сообщение настоящим открытым ключом Джона и переслать ему. Джон вряд ли догадается о том, что на самом деле произошло.

Звучит пугающе? Еще бы. Ну и каково же решение? Решение заключается в том, чтобы как-то проверить подлинность открытого ключа и убедиться, что он создан настоящим отправителем. Для этого можно воспользоваться проверкой *отпечатков пальцев*. Эта тема выходит за пределы данной книги, но, в двух словах, когда Джейн шифрует сообщение открытым ключом, она проверяет «отпечаток пальца» – сигнатуру ключа, убеждаясь в том, что он действительно принадлежит Джону. (Это обсуждение показывает также необходимость защиты канала связи между отправителем и получателем.)

Данные зашифровываются и расшифровываются разными ключами, так как же расшифровывающий процесс узнает, какой ключ использовался при шифровании? Вспомните, что оба ключа были сгенерированы получателем одновременно, вследствие чего между ними существует определенная математическая зависимость. Один из них – просто противоположность другого: сделанное с помощью одного ключа можно отменить при помощи другого. Поэтому расшифровывающий процесс может восстановить данные без использования ключа шифрования.

Вследствие того, что между открытым и секретным ключами существует математическая зависимость, теоретически можно вычислить секретный ключ на основе открытого ключа, хотя это весьма трудоем-

кий процесс, требующий разложения на множители очень больших чисел. Поэтому для уменьшения вероятности взлома методом «грубой силы» обычно используются ключи длиной 1024 бит, в то время как при симметричном шифровании применяются 56-, 64-, 128- и 256-битовые ключи. Длина в 1024 бита является типовой, но не стандартной. Более короткие ключи также могут быть использованы.

Oracle позволяет использовать асимметричное шифрование в двух ситуациях:

- При передаче данных между клиентом и сервером.
- При аутентификации пользователей.

В обоих случаях требуется наличие компонента Oracle Advanced Security Option (ASO), отсутствующего в базовой конфигурации и поставляемого за отдельную плату. Этот компонент активирует шифрование асимметричным ключом только в указанных случаях; он не имеет простого, готового к использованию интерфейса, который позволил бы организовать шифрование всех хранимых данных.

Единственный ориентированный на разработчиков инструмент, свободно доступный в Oracle, обеспечивает симметричное шифрование. Поэтому в данной главе мы сосредоточим внимание на этом виде шифрования.

Так как при асимметричном шифровании используются разные ключи для зашифровывания и расшифровывания, отправителю не надо знать секретный ключ, который будет использован получателем. При симметричном шифровании, напротив, используется один и тот же ключ, поэтому в таких системах защита ключей очень важна.

Алгоритмы шифрования

Существует много распространенных и коммерчески доступных алгоритмов шифрования, но мы ограничимся здесь рассмотрением алгоритмов шифрования симметричным ключом, поддерживаемых Oracle в PL/SQL-программах. Алгоритмы DES и Triple DES поддерживаются обоими встроенными пакетами Oracle: `DBMS_CRYPTO` и `DBMS_OBFUSCATION_TOOLKIT`; но только в `DBMS_CRYPTO`, появившемся в версии Oracle 10*g* Release 1, поддерживается алгоритм AES.

Data Encryption Standard (DES)

Исторически господствующим стандартом шифрования стал алгоритм DES. Он был разработан более 20 лет назад для Национального бюро стандартов США (National Bureau of Standards), позднее превратившегося в Национальный институт стандартов и технологии (National Institute of Standards and Technology, NIST). Вследствие этого DES стал стандартом Американского национального института стандартов (American National Standards Institute, ANSI).

Об алгоритме DES и его истории можно рассказывать очень долго, но здесь мы не станем его описывать, а рассмотрим лишь его применение в БД Oracle. Этот алгоритм работает с 64-разрядными ключами, но отбрасывает 8 разрядов, фактически используя только 56. Взломщику надо перебрать 72 057 594 037 927 936 комбинаций, чтобы угадать ключ.

Некоторое время DES вполне соответствовал требованиям, но, в конце концов, начал устаревать. Для современных мощных компьютеров его взлом не представляет трудности, несмотря на необходимость перебора большого количества комбинаций при поиске ключа.

Triple DES (DES3)

Институт NIST предложил разработать на основе исходного алгоритма DES новую схему, в которой шифрование данных выполнялось бы дважды или трижды, в зависимости от используемого режима. Злоумышленник, пытающийся взломать ключ, должен будет перебрать 2^{112} и 2^{168} комбинаций в двух- и трехпроходном режиме шифрования соответственно. В DES3 используется ключ длиной 128 или 192 бит, в зависимости от использования двух- или трехпроходной схемы.

Алгоритм Triple DES тоже уже начинает устаревать и, как и DES, становится ненадежным при определенном виде атак.

Advanced Encryption Standard (AES)

В ноябре 2001 г. в 197-м выпуске FIPS (Federal Information Processing Standards – Федеральные стандарты обработки информации) было объявлено об утверждении нового стандарта AES (Advanced Encryption Standard – улучшенный стандарт шифрования), вступающего в силу с мая 2002 года. Полный текст стандарта находится на сайте NIST по адресу *http://csrc.nist.gov/CryptoToolkit/aes/round2/r2report.pdf* (эта ссылка имеется на сайте нашей книги).

Ниже в этой главе мы покажем, как использовать перечисленные алгоритмы, указывая соответствующие параметры или константы во встроенных пакетах Oracle.

Дополнение и сцепление

В процессе шифрования фрагмент данных не обрабатывается целиком. Обычно он разбивается на блоки по 8 байт, каждый из которых обрабатывается независимо. Конечно, длина такого блока не обязательно кратна восьми; в таком случае алгоритм добавляет недостающие символы в последнюю порцию, удлиняя ее ровно до 8 байт. Такой процесс называется *дополнением*. Дополнение должно производиться так, чтобы взломщик не мог определить, что именно было добавлено, и использовать эти знания для взлома ключа шифрования. Чтобы введение заполнителя не влияло на безопасность, можно воспользоваться доступным в Oracle готовым методом дополнения Public Key Crypto-

graphy System #5 (PKCS#5). Есть еще несколько режимов, позволяющих выполнять дополнение нолями или не использовать заполнители вовсе. Позже в этой главе мы покажем, как использовать заполнители, указывая параметры или выбирая константы встроенных пакетов Oracle.

Так как данные делятся на блоки, необходим способ соединения этих блоков обратно. Этот процесс называется *сцеплением*. Надежность системы шифрования в целом зависит от того, как соединяются и зашифровываются блоки – по отдельности или совместно с соседними блоками. Oracle поддерживает следующие методы сцепления:

CBC

Cipher Block Chaining – сцепление блоков шифротекста, наиболее распространенный метод сцепления.

ECB

Electronic Code Book – электронная кодовая книга.

CFB

Cipher Feedback – обратная связь по шифротексту.

OFB

Output Feedback – обратная связь по выводу.

Ниже мы рассмотрим, как использовать эти методы, выбирая параметры и константы во встроенных пакетах Oracle.

Шифрование в Oracle9i

Давайте начнем подробное обсуждение шифрования в Oracle с обзора пакета `DBMS_OBFUSCATION_TOOLKIT`. Несмотря на то что сейчас Oracle рекомендует пользоваться более новым пакетом `DBMS_CRYPTO`, большинство организаций еще не перевели на него свои приложения, поэтому есть смысл начать с более старого пакета.

Если вы работаете с Oracle 10*g* и начинаете новый проект, то, вероятно, захотите использовать возможности, описанные в разделе «Шифрование в Oracle 10*g*». Однако чтобы уверенно ориентироваться в основных понятиях шифрования, вам, возможно, будет полезно прочитать сначала этот раздел.

Шифрование данных

Пора посмотреть на шифрование в Oracle в действии. Приведем простой пример, а затем рассмотрим его подробно. Предположим, вы хотите получить зашифрованное представление строки «`SHHH..TOP SECRET`». Это делается следующим фрагментом кода, в котором вызывается процедура `DES3ENCRYPT` пакета `DBMS_OBSFUSCATION_TOOLKIT`:

```
1  DECLARE
2     l_enc_val   VARCHAR2 (200);
3  BEGIN
4     DBMS_OBFUSCATION_TOOLKIT.des3encrypt
5                  (input_string          => 'SHHH..TOP SECRET',
6                  key_string            => 'ABCDEFGHIJKLMNOP',
7                  encrypted_string      => l_enc_val
8                  );
9                   DBMS_OUTPUT.put_line ('Encrypted Value = ' || l_enc_val);
10    END;
```

Результат выглядит так:

```
Encrypted Value = ¿jVªå¬F.(e) ?«?0
```

В строке 6 указан ключ, используемый для шифрования исходного значения, длина этого ключа равна 16 символам. Зашифрованное значение имеет тип VARCHAR2, но содержит управляющие символы. В таком виде результат мало полезен в реальных приложениях, особенно если вы собираетесь его хранить, распечатывать или сообщать кому-нибудь; вероятно, имеет смысл сделать его более удобным, преобразовав к печатному виду. Заметьте, однако, что приведение данных к типу RAW и обратно не всегда желательно; ниже это обсуждается во врезке «Когда следует использовать шифрование в формате RAW?». Наша первая задача заключается в преобразовании к типу данных RAW при помощи встроенного пакета UTL_RAW.

```
l_enc_val := utl_raw.cast_to_raw(l_enc_val);
```

Затем преобразуем полученное значение функцией RAWTOHEX, чтобы с ним было легче работать:

```
l_enc_val := rawtohex(utl_raw.cast_to_raw(l_enc_val));
```

Теперь наш PL/SQL-блок выглядит так:

```
DECLARE
   l_enc_val   VARCHAR2 (200);
BEGIN
   DBMS_OBFUSCATION_TOOLKIT.des3encrypt (input_string   => 'SHHH..TOP SECRET',
                                         key_string     => 'ABCDEFGHIJKLMNOP',
                                         encrypted_string => l_enc_val
                                        );
   l_enc_val := RAWTOHEX (UTL_RAW.cast_to_raw (l_enc_val));
   DBMS_OUTPUT.put_line ('Encrypted Value = ' || l_enc_val);
END;
```

Получим такой результат:

```
Encrypted Value = A86A56A6EE92462E28652903ECAEC730
```

Результат представлен в виде шестнадцатеричной строки, которую удобно хранить и обрабатывать в поле таблицы с типом VARCHAR2. Можно преобразовать результат к десятичному виду, что удобно для число-

вой обработки, но, как правило, лучше оставить его в виде строки шестнадцатеричных символов – будет понятно, что это зашифрованные данные.

```
l_enc_val := to_number('A86A56A6EE92462E28652903ECAEC730',
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX')

223862444271805716712258987042708309808
```

Основываясь на программах шифрования из пакета DBMS_OBFUSCATION_TOOLKIT, напишем несколько функций-оберток, чтобы сделать процесс шифрования более простым и гибким.

В этом примере вызывается процедура DES3ENCRYPT, выполняющая шифрование по алгоритму Triple DES. Семейство ENCRYPT включает в себя ряд других процедур и функций, полные спецификации которых приведены в приложении A.

```
/* Файл на веб-сайте: get_enc_val_1.sql */
CREATE OR REPLACE FUNCTION get_enc_val (p_in_val IN VARCHAR2,
                                        p_key IN VARCHAR2)
   RETURN VARCHAR2
IS
   l_enc_val   VARCHAR2 (200);
BEGIN
   l_enc_val :=
DBMS_OBFUSCATION_TOOLKIT.des3encrypt (input_string      => p_in_val,
                                      key_string        => p_key
                                          );
   l_enc_val := RAWTOHEX (UTL_RAW.cast_to_raw (l_enc_val));
   RETURN l_enc_val;
END;
```

Использовав эту функцию в предыдущем примере, получим искомый результат.

```
SQL> SET SERVEROUTPUT ON

SQL> DECLARE
 2>    v_enc   VARCHAR2 (200);
 3>  BEGIN
 4>    v_enc := get_enc_val ('SHHH..TOP SECRET', 'ABCDEFGHIJKLMNOP');
 5>    DBMS_OUTPUT.put_line ('Encrypted value = ' || v_enc);
 6>  END;
 7>  /

Encrypted value = A86A56A6EE92462E28652903ECAEC730

PL/SQL procedure successfully completed.
```

Конечно, реальное значение, полученное в вашей системе, может быть другим вследствие использования другого набора символов; это очень важный момент, к которому мы еще вернемся. Данную функцию шифрования вы можете использовать разными способами – для встав-

ки данных в зашифрованные столбцы, передачи зашифрованных данных в другие процедуры и функции и т. п.

Прежде чем двигаться дальше, протестируем нашу функцию на разных входных значениях. В первом примере мы зашифровывали строку «SHHH. . TOP SECRET». Теперь зашифруем другое значение:

```
DECLARE
   v_enc   VARCHAR2 (200);
BEGIN
   v_enc := get_enc_val ('A DIFFERENT VALUE', 'ABCDEFGHIJKLMNOP');
   DBMS_OUTPUT.put_line ('Encrypted value = ' || v_enc);
END;
/
```

Что-то не так. На этот раз возникает ошибка.

```
DECLARE
*
ERROR at line 1:
ORA-28232: invalid input length for obfuscation toolkit
ORA-06512: at "SYS.DBMS_OBFUSCATION_TOOLKIT_FFI", line 0
ORA-06512: at "SYS.DBMS_OBFUSCATION_TOOLKIT", line 216
ORA-06512: at "SCOTT.GET_ENC_VAL", line 10
ORA-06512: at line 4
```

Что здесь неправильно? Единственное, что изменилось, – это входная строка: в первый раз она имела длину 16 символов, а теперь мы передаем 17. Оказывается, длина входной строки в процедуре DES3ENCRYPT должна быть в точности кратна восьми; если это не так, возникает исключение с кодом ошибки ORA-28232. При данном типе шифрования, известном как *блочное шифрование*, программа обрабатывает данные поблочно (а блок состоит из восьми символов). Если длина входного значения не кратна восьми, строка должна быть дополнена до этой длины, как уже говорилось в разделе «Дополнение и сцепление». Внутри функции можно легко привести входную строку к требуемой длине:

```
/* Файл на веб-сайте: get_enc_val_2.sql */
CREATE OR REPLACE FUNCTION get_enc_val (p_in_val IN VARCHAR2,
                                        p_key IN VARCHAR2)
   RETURN VARCHAR2
IS
   l_enc_val   VARCHAR2 (200);
   l_in_val    VARCHAR2 (200);
BEGIN
   l_in_val := RPAD (p_in_val, (8 * ROUND (LENGTH (p_in_val) / 8, 0) + 8));
   l_enc_val :=
      DBMS_OBFUSCATION_TOOLKIT.des3encrypt (input_string    => l_in_val,
                                            key_string      => p_key
                                           );
   l_enc_val := RAWTOHEX (UTL_RAW.cast_to_raw (l_enc_val));
   RETURN l_enc_val;
```

```
END;
/
```

Единственное отличие этого варианта функции в том, что входная строка дополняется справа пробелами до длины, кратной восьми. С помощью этой функции вы можете зашифровывать строки любой длины.

Если вы используете пакет DBMS_CRYPTO в Oracle 10*g*, вам не надо самим выравнивать входное значение; дополнение осуществляется в пакете. Как уже говорилось, в этом пакете расширен выбор алгоритмов шифрования и методов дополнения и сцепления.

Задание вектора инициализации

Функция шифрования, описанная в предыдущем разделе, отлично работает в большинстве ситуаций. Но злоумышленники все еще на шаг впереди. Один из используемых ими инструментов для взлома (называемого также *криптоанализ*) предназначен для изучения заголовка зашифрованных данных и идентификации заполнителя. Вы можете противодействовать этому, вставив перед началом данных некое случайное число, не имеющее к ним отношения. Это что-то вроде вашего собственного простого шифра. Например, если реальные данные представлены последовательностью 12345678, вы можете предварить их случайным значением, скажем 6675, получив значение 667512345678, которое затем будет зашифровано. В результате заголовок будет содержать данные, относящиеся к числу 6675, а не к действительным данным. При расшифровывании вам надо не забыть удалить эту случайную величину.

Эта случайная последовательность, вставляемая перед началом данных, называется *вектором инициализации* (Initialization Vector – IV). В пакете DBMS_OBFUSCATION_TOOLKIT вектор инициализации передается процедуре DES3ENCRYPT в дополнительном параметре iv_string. Т. к. вектор инициализации присоединяется к действительным данным, длину, кратную восьми, должна иметь результирующая строка, а не исходная. Давайте изменим нашу функцию шифрования так, чтобы она принимала этот параметр и увеличивала длину до значения, кратного восьми.

```
/* Файл на веб-сайте get_enc_val_3.sql */
CREATE OR REPLACE FUNCTION get_enc_val (
   p_in_val   IN   VARCHAR2,
   p_key      IN   VARCHAR2,
   p_iv       IN   VARCHAR2 := NULL
)
   RETURN VARCHAR2
IS
   l_enc_val   VARCHAR2 (200);
   l_in_val    VARCHAR2 (200);
   l_iv        VARCHAR2 (200);
BEGIN
```

```
   l_in_val := RPAD (p_in_val, (8 * ROUND (LENGTH (p_in_val) / 8, 0) + 8));
   l_iv := RPAD (p_iv, (8 * ROUND (LENGTH (p_iv) / 8, 0) + 8));
   l_enc_val :=
      DBMS_OBFUSCATION_TOOLKIT.des3encrypt (input_string    => l_in_val,
                                            key_string      => p_key,
                                            iv_string       => l_iv
                                           );
   l_enc_val := RAWTOHEX (UTL_RAW.cast_to_raw (l_enc_val));
   RETURN l_enc_val;
END;
```

Расшифровывание данных

До сих пор мы только зашифровывали данные, теперь давайте посмотрим, как расшифровать их процедурой DES3DECRYPT и получить исходное значение. В приведенном блоке PL/SQL зашифровывается *открытый текст*, а затем расшифровывается полученное значение.

```
DECLARE
   l_enc_val   VARCHAR2 (2000);
   l_dec_val   VARCHAR2 (2000) := 'Clear Text Data';
   l_key       VARCHAR2 (2000) := 'ABCDEFGHIJKLMNOP';
BEGIN
   l_enc_val := get_enc_val (l_dec_val, l_key, '12345678');
   l_dec_val :=
      DBMS_OBFUSCATION_TOOLKIT.des3decrypt
             (input_string      => UTL_RAW.cast_to_varchar2
                                                  (HEXTORAW (l_enc_val)
                                                  ),
              key_string        => l_key
             );
   DBMS_OUTPUT.put_line ('Decrypted Value = ' || l_dec_val);
END;
/
```

Результат выглядит так:

```
Decrypted Value = s}??2+  ¬xt Data

PL/SQL procedure successfully completed.
```

Но постойте! Расшифрованное значение *отличается* от того, которое было зашифровано. Где же ошибка?

Посмотрите на параметры процедуры DES3DECRYPT. Вы передали ей вектор инициализации? Так как вектор инициализации был указан при зашифровывании, он должен быть также указан и при расшифровывании. Перепишем этот блок, используя вектор инициализации 12345678:

```
DECLARE
   l_enc_val   VARCHAR2 (2000);
   l_dec_val   VARCHAR2 (2000) := 'Clear Text Data';
   l_key       VARCHAR2 (2000) := 'ABCDEFGHIJKLMNOP';
BEGIN
```

```
   l_enc_val := get_enc_val (l_dec_val, l_key, '12345678');
   l_dec_val :=
      DBMS_OBFUSCATION_TOOLKIT.des3decrypt
              (input_string      => UTL_RAW.cast_to_varchar2
                                                    (HEXTORAW (l_enc_val)
                                                   ),
               key_string        => l_key,
               iv_string         => '12345678'
              );
   DBMS_OUTPUT.put_line ('Decrypted Value = ' || l_dec_val);
END;
/
```

Получим ожидаемый результат:

```
Decrypted Value = Clear Text Data

PL/SQL procedure successfully completed.
```

Если при зашифровывании вы используете вектор инициализации, то вы должны указать *тот же самый* вектор при расшифровывании.

В некотором смысле вектор инициализации выступает в качестве ключа или его части, но он не может заменить собой ключ как таковой. Почему? Рассмотрим следующий пример.

```
DECLARE
   l_enc_val   VARCHAR2 (2000);
   l_dec_val   VARCHAR2 (2000) := 'Clear Text Data';
   l_key       VARCHAR2 (2000) := 'ABCDEFGHIJKLMNOP';
BEGIN
   l_enc_val := get_enc_val (l_dec_val, l_key, '12345678');
   l_dec_val :=
      DBMS_OBFUSCATION_TOOLKIT.des3decrypt
              (input_string      => UTL_RAW.cast_to_varchar2
                                                    (HEXTORAW (l_enc_val)
                                                   ),
               key_string        => l_key,
               iv_string         => '1234567X'
              );
   DBMS_OUTPUT.put_line ('Decrypted Value = ' || l_dec_val);
END;
/
```

В результате выполнения получим:

```
Decrypted Value = Clear T?xt Data

PL/SQL procedure successfully completed.
```

Данные были зашифрованы с вектором инициализации `12345678`, а расшифрованы с вектором `1234567X`, отличающимся только восьмым символом. Из-за этого расшифрованное значение отличается от исходного: в восьмой позиции вместо буквы `e` появился непечатаемый символ.

Несмотря на отличие полученного текста от оригинала, его угадывание облегчается за счет перебора значений вектора инициализации; данная процедура называется *атака методом грубой силы*. В силу того, что векторы инициализации обычно короче, чем ключи, такое угадывание займет намного меньше времени. Поэтому не стоит полагаться на использование этого вектора в качестве ключа.

Вектор инициализации всего лишь изменяет шифруемый открытый текст во избежание повторов; он не может служить заменой ключу шифрования.

Шифрование данных типа RAW

Мы уже касались темы использования данных типа `RAW`. Теперь разберемся, как шифровать данные такого типа, воспользовавшись тем фактом, что процедуры `DES3ENCRYPT` и `DES3DECRYPT` пакета `DBMS_OBFUSCATION_TOOLKIT` перегружены. Это означает, что они существуют в нескольких вариантах. Каждый вариант содержит процедуры с одинаковым набором входных параметров и возвращаемым значением `encrypted_string` или `decrypted_string` (в зависимости от того, зашифровываете вы или расшифровываете). Существуют также версии этих функций и процедур, которые могут работать с данными типа `RAW`. Эти версии можно использовать для обработки «сырых» данных, таких как большие объекты (LOB).

Разумеется, данные при шифровании можно привести к типу `RAW`, как показано в этом примере:

```
/* Файл на веб-сайте: enc_raw.sql */
CREATE OR REPLACE FUNCTION enc_raw (
   p_in_val   IN   VARCHAR2,
   p_key      IN   VARCHAR2,
   p_iv       IN   VARCHAR2
)
   RETURN VARCHAR2
IS
   l_enc_val   RAW (200);
   l_in_val    RAW (200);
   l_iv        RAW (200);
BEGIN
   l_in_val :=
      UTL_RAW.cast_to_raw (RPAD (p_in_val,
                                 (8 * ROUND (LENGTH (p_in_val) / 8, 0) + 8
                                 )
                                )
                           );
   l_iv :=
       UTL_RAW.cast_to_raw (RPAD (p_iv, (8 * ROUND (LENGTH (p_iv) / 8, 0) +
                                                                  + 8)));
   l_enc_val :=
      DBMS_OBFUSCATION_TOOLKIT.des3encrypt (input      => l_in_val,
                                            KEY        => p_key,
```

```
                                          iv          => l_iv
                                         );
   RETURN RAWTOHEX (UTL_RAW.cast_to_raw (l_enc_val));
END;
/
```

Однако преобразование из типа VARCHAR2 в тип RAW требует дополнительных затрат, которые могут заметно *снизить* производительность. В проведенных тестах показанная здесь версия для типов VARCHAR2 и NUMBER оказалась примерно на 50% медленнее версии, работающей с простыми строками. Т. к. процесс шифрования активно использует процессор, в других системах результат может заметно отличаться. Однако общее правило заключается в том, чтобы по возможности избегать манипуляций с типом RAW, если изначально известно, что ваши данные имеют символьный тип, и вы используете только один набор символов.

Многопроходное шифрование

В этой главе (в разделе «Алгоритмы шифрования») я уже упоминал об усовершенствовании стандарта DES за счет *двойного* или *тройного* шифрования содержимого, в результате чего и появилось название *Triple DES* или DES3. В Oracle DES3 реализуется в функции DES3ENCRYPT, которая по умолчанию использует двухпроходное шифрование. Однако, используя новый параметр which, вы можете указать функции на необходимость выполнения трех проходов. Значение по умолчанию (0) соответствует двум проходам, а значение 1 – трем. Естественно, три прохода обеспечивают большую надежность.

Для использования трехпроходного алгоритма необходим ключ длиною не меньше 24 байт (вместо 16 байт, которыми мы довольствовались до сих пор). Изменим исходную функцию, с тем чтобы позволить пользователю выбрать количество проходов: 2 или 3.

Когда следует использовать шифрование в формате RAW?

Во-первых, как уже говорилось, шифрование в формате RAW используется для данных типа BLOB.

Во-вторых, шифрование в формате RAW используется в случае, если в базе данных используются буквы не английского алфавита. Если вы пользуетесь функциональностью *Oracle Globalization Support* (которая раньше называлась *National Language Support* – NLS), то шифрование в формате RAW обеспечит обработку таких символов без выполнения каких-либо дополнительных операций (в частности, при экспорте и импорте данных). Зашифрованные данные можно будет перемещать из одной базы данных в другую, не опасаясь их повреждения.

```
/* Файл на веб-сайте: get_enc_val_4.sql */
CREATE OR REPLACE FUNCTION get_enc_val (
   p_in_val   IN   VARCHAR2,
   p_key      IN   VARCHAR2,
   p_iv       IN   VARCHAR2,
   p_which    IN   NUMBER := 0
)
   RETURN VARCHAR2
IS
   l_enc_val   VARCHAR2 (200);
   l_in_val    VARCHAR2 (200);
   l_iv        VARCHAR2 (200);
BEGIN
   l_in_val := RPAD (p_in_val, (8 * ROUND (LENGTH (p_in_val) / 8, 0) + 8));
   l_iv := RPAD (p_iv, (8 * ROUND (LENGTH (p_iv) / 8, 0) + 8));
   l_enc_val :=
      DBMS_OBFUSCATION_TOOLKIT.des3encrypt (input_string      => l_in_val,
                                            key_string        => p_key,
                                            iv_string         => l_iv,
                                            which             => p_which
                                           );
   l_enc_val := RAWTOHEX (UTL_RAW.cast_to_raw (l_enc_val));
   RETURN l_enc_val;
END;
/
```

Изменение количества проходов при зашифровывании означает также, что и расшифровывание должно быть трехпроходным. При расшифровывании необходимо явно установить параметр `which` **в значение** `1`**.**

```
DECLARE
   l_enc_val   VARCHAR2 (2000);
   l_dec_val   VARCHAR2 (2000) := 'Clear Text Data';
   l_key       VARCHAR2 (2000) := 'ABCDEFGHIJKLMNOPQRSTUVWXY';
BEGIN
   l_enc_val := get_enc_val (l_dec_val, l_key, '12345678', 1);
   l_dec_val :=
      DBMS_OBFUSCATION_TOOLKIT.des3decrypt
              (input_string      => UTL_RAW.cast_to_varchar2
                                                    (HEXTORAW (l_enc_val)
                                                       ),
               key_string        => l_key,
               iv_string         => '12345678',
               which             => 1
              );
   DBMS_OUTPUT.put_line ('Decrypted Value = ' || l_dec_val);
END;
/
```

Длина ключа теперь составляет 24 байта – минимум, необходимый для выполнения трехпроходного шифрования.

Сводим воедино

Мы изучили составляющие процесса шифрования, а теперь давайте попробуем соединить их все вместе и создать наш собственный комплексный инструмент шифрования. Изменим нашу старую добрую функцию get_enc_val следующим образом:

```
/* Файл на веб-сайте: get_enc_val_5.sql */
CREATE OR REPLACE FUNCTION get_enc_val (
   p_in_val   IN   VARCHAR2,
   p_key      IN   VARCHAR2,
   p_iv       IN   VARCHAR2 := NULL,
   p_which    IN   NUMBER := 0
)
   RETURN VARCHAR2
IS
   l_enc_val   VARCHAR2 (200);
   l_in_val    VARCHAR2 (200);
   l_iv        VARCHAR2 (200);
BEGIN
   IF p_which = 0
   THEN
      IF LENGTH (p_key) < 16
      THEN
         raise_application_error
                           (-20001,
                            'Key length less than 16 for two-pass scheme'
                           );
      END IF;
   ELSIF p_which = 1
   THEN
      IF LENGTH (p_key) < 24
      THEN
         raise_application_error
                          (-20002,
                           'Key length less than 24 for three-pass scheme'
                          );
      END IF;
   ELSE
      raise_application_error (-20003,
                               'Incorrect value of which '
                            || p_which
                            || '; must be 0 or 1'
                           );
   END IF;

   l_in_val := RPAD (p_in_val, (8 * ROUND (LENGTH (p_in_val) / 8, 0) + 8));
   l_iv := RPAD (p_iv, (8 * ROUND (LENGTH (p_iv) / 8, 0) + 8));
   l_enc_val :=
      DBMS_OBFUSCATION_TOOLKIT.des3encrypt (input_string      => l_in_val,
                                            key_string        => p_key,
                                            iv_string         => l_iv,
                                            which             => p_which
```

```
                                                    );
   l_enc_val := RAWTOHEX (UTL_RAW.cast_to_raw (l_enc_val));
   RETURN l_enc_val;
END;
/
```

Аналогичную функцию создадим для расшифровывания и назовем ее get_dec_val.

```
/* Файл на веб-сайте: get_dec_val_1.sql */
CREATE OR REPLACE FUNCTION get_dec_val (
   p_in_val   VARCHAR2,
   p_key      VARCHAR2,
   p_iv       VARCHAR2 := NULL,
   p_which    NUMBER := 0
)
   RETURN VARCHAR2
IS
   l_dec_val   VARCHAR2 (2000);
   l_iv        VARCHAR2 (2000);
BEGIN
   IF p_which = 0
   THEN
      IF LENGTH (p_key) < 16
      THEN
         raise_application_error
                              (-20001,
                               'Key length less than 16 for two-pass scheme'
                              );
      END IF;
   ELSIF p_which = 1
   THEN
      IF LENGTH (p_key) < 24
      THEN
         raise_application_error
                             (-20002,
                              'Key length less than 24 for three-pass scheme'
                             );
      END IF;
   ELSE
      raise_application_error (-20003,
                                  'Incorrect value of which '
                               || p_which
                               || '; must be 0 or 1'
                              );
   END IF;

   l_iv := RPAD (p_iv, (8 * ROUND (LENGTH (p_iv) / 8, 0) + 8));
   l_dec_val :=
      DBMS_OBFUSCATION_TOOLKIT.des3decrypt
                (input_string      => UTL_RAW.cast_to_varchar2
                                                    (HEXTORAW (p_in_val)
                                                    ),
```

```
                key_string        => p_key,
                iv_string         => l_iv,
                which             => p_which
               );
   RETURN RTRIM (l_dec_val);
END;
/
```

Обратите внимание, что при зашифровывании я дополнил параметр, содержащий вектор инициализации, пробелами так, чтобы его длина была кратна восьми. После расшифровывания необходимо удалить эти добавленные пустые символы, как мы сделали это в строке возврата приведенной выше функции.

Невозможно зашифровать данные, уже зашифрованные средствами пакета DBMS_OBFUSCATION_TOOLKIT. При подобной попытке пакет генерирует ошибку ORA-28233 в связи с невозможностью двойного шифрования.

Генерирование ключей

Из приведенных выше рассуждений должно быть очевидно, что самым слабым местом шифрования является его ключ. Для успешной расшифровки зашифрованных данных необходим ключ (он в прямом смысле является *ключом* к успеху), поэтому для обеспечения безопасности шифрования необходимо сделать угадывание ключа чрезвычай-

Основы шифрования в Oracle9i

- Шифрование может выполняться по алгоритму DES или DES3 (Triple DES), причем DES3 является предпочтительным.
- Шифрование DES3 может быть двух- или трехпроходным. По умолчанию используются два прохода.
- Длина шифруемой строки должна быть кратна восьми.
- Ключ, используемый для зашифровывания, должен использоваться и при расшифровывании.
- Длина ключа должна быть не меньше 16 для DES и двухпроходного DES3, и не меньше 24 для трехпроходного DES3.
- Для усиления защиты данных возможно использование вектора инициализации Если вектор инициализации использован при зашифровывании, его также необходимо указать при расшифровывании.
- Так как вектор инициализации присоединяется в начало входного значения, длина полученной объединенной строки должна быть кратна восьми.

но сложным. В рассмотренных ранее примерах использовался 16-байтный ключ для двухпроходного шифрования DES3 и 24-байтный ключ для трехпроходного шифрования DES3.

При выборе ключа шифрования помните о двух важных моментах:

- Чем длиннее ключ, тем сложнее его угадать. Двухпроходный метод допускает использование ключа из 128 бит, а трехпроходный – из 192 бит. Следует выбирать наиболее длинный из возможных ключей.
- Ключ должен быть не только длинным, он не должен соответствовать никакому шаблону, который было бы легко угадать. Например, ранее в примере я использовал в качестве ключа последовательность цифр во вполне предсказуемом порядке – `1234567890123456`. Это недопустимо. Значение `a2H8s7X40Ys8346yp2` гораздо более удачно.

Использование DES3GETKEY

В пакет `DBMS_OBFUSCATION_TOOLKIT` включена функция `DES3GETKEY` (а также процедура, причем оба формата перегружены с несколькими типами данных), которая позволяет сгенерировать криптографически допустимый ключ. Функции для генерирования случайного значения, которое может быть использовано в качестве ключа, необходимо передать значение для инициализации.

Пакет `DBMS_CRYPTO`, доступный в Oracle 10*g*, содержит функцию `GETRANDOMBYTES`, которая может использоваться для формирования криптографически случайных ключей.

Функция вызывается следующим образом:

```
l_ret := DBMS_OBFUSCATION_TOOLKIT.des3getkey (
   seed_string => l_seed
);
```

Значение переменной `l_seed` – случайная строка длиной 80 символов (более длинное значение будет принято, но использованы будут только 80 символов). Возвращаемое значение имеет тип `VARCHAR2` и записывается в переменную `l_ret`. Длина начального значения должна быть равна 80 символам, поэтому для генерирования значения используем простой алгоритм. (Помните, что сейчас мы генерируем не ключ, а только значение для инициализации генератора случайных чисел. Более подробно это обсуждается в главе 7.)

```
l_seed varchar2(2000) :=
   '1234567890'||
   '1234567890'||
   '1234567890'||
   '1234567890'||
   '1234567890'||
   '1234567890'||
```

```
    '1234567890'||
    '1234567890'
```

Функция DES3GETKEY возвращает значение в двоичном формате, которое, вероятно, должно быть преобразовано к пригодному для употребления типу данных (например, VARCHAR2), так что я могу изменить возвращаемый ключ следующим образом:

```
l_ret := rawtohex(utl_raw.cast_to_raw(l_ret));
```

Ключ преобразуется в значение типа RAW, а затем в шестнадцатеричное значение. Еще один параметр – which – используется для указания того, два или три прохода будет использоваться при шифровании.

Собрав все воедино, получим такую функцию генерирования ключа.

```
/* Файл на веб-сайте: get_key_1.sql */
  1  CREATE OR REPLACE FUNCTION get_key (
  2     p_seed    VARCHAR2 :=    '1234567890'
  3                            || '1234567890'
  4                            || '1234567890'
  5                            || '1234567890'
  6                            || '1234567890'
  7                            || '1234567890'
  8                            || '1234567890'
  9                            || '1234567890',
 10     p_which   NUMBER := 0
 11  )
 12     RETURN VARCHAR2
 13  IS
 14     l_seed   VARCHAR2 (80);
 15     l_ret    VARCHAR2 (2000);
 16  BEGIN
 17     l_seed := RPAD (p_seed, 80);
 18     l_ret :=
 19        DBMS_OBFUSCATION_TOOLKIT.des3getkey (seed_string      => l_seed,
 20                                             which            => p_which
 21                                            );
 22     l_ret := RAWTOHEX (UTL_RAW.cast_to_raw (l_ret));
 23     RETURN l_ret;
 24* END;
```

Важные элементы кода функции пояснены в таблице:

Строки	Описание
2–9	Чрезвычайно важным параметром является значение, инициализирующее последовательность случайных чисел, которое по умолчанию равно восьмикратно повторенной (для образования 80-байтной строки) строке 1234567890. Естественно, это ненадежно, так что используйте вместо этого значения произвольную 80-байтную строковую константу. Более длинные строки не улучшат «случайность», так как использованы будут только первые 80 символов.

Строки	Описание
10	По умолчанию предполагается, что будет сгенерирован ключ DES3 для двухпроходного шифрования, т. е. параметр which установлен в 0. Для трехпроходного шифрования необходимо установить данный параметр в значение 1.
17	Начальное значение должно иметь длину 80 байт. Если пользователь предоставляет более короткое значение, то функция примет его и дополнит до 80 байт, вместо того чтобы генерировать ошибку.
18	Функция DES3GETKEY возвращает значение типа VARCHAR2.
22	Возвращаемое значение имеет тип VARCHAR2, но содержит множество управляющих символов. Преобразуем его сначала к типу данных RAW, а затем – к шестнадцатеричному значению.

Эта функция при каждом вызове будет возвращать криптографически случайное значение. Давайте посмотрим, как это работает:

```
BEGIN
   DBMS_OUTPUT.put_line ('Key=' || get_key);
   DBMS_OUTPUT.put_line ('Key=' || get_key);
   DBMS_OUTPUT.put_line ('Key=' || get_key);
   DBMS_OUTPUT.put_line ('Key=' || get_key);
   DBMS_OUTPUT.put_line ('Key=' || get_key);
END;
```

Результат будет таким:

```
Key=4992D7CCC6B9428F11D7EC612E728C02
Key=4DB67B0610E3EB2EB6B7B6B39DC4DB13
Key=4DC1F80A3FE4FC266A667CE2A11E25C9
Key=111768ECC7E6F0C5DFAD6B9B0C146C9A
Key=75FE17395B8209FC578C41B26E22CBC7
```

Обратите внимание, что генерируемый ключ каждый раз будет разным, несмотря на то, что начальное значение последовательности случайных чисел не изменится. Поэтому когда вы запустите функцию, реальное возвращенное значение может быть другим; можно считать его случайным.

Использование ключа в шифровании

Используя только что созданную функцию, можно достаточно хорошо зашифровать секретные данные. Рассмотрим простой пример:

```
DECLARE
   l_key   VARCHAR2 (80);
   l_enc   VARCHAR2 (2000);
BEGIN
   l_key := get_key;
   l_enc := get_enc_val ('Input Value', l_key);
   DBMS_OUTPUT.put_line ('Key = ' || l_key || ' Encrypted Value = ' || l_enc);
END;
/
```

Формирование ключей в более ранних версиях Oracle

К сожалению, функция DES3GETKEY недоступна в версии Oracle8*i*. Вам придется создать собственный генератор случайных строк для получения подходящего ключа. Здесь пригодятся наши знания о генерации случайных строк (см. главу 7). Итак, в ранних версиях можно создать свою собственную функцию get_key следующим образом:

```
1  CREATE OR REPLACE FUNCTION get_key
2     RETURN VARCHAR2
3  IS
4     l_ret   VARCHAR2 (200);
5  BEGIN
6     l_ret := DBMS_RANDOM.STRING ('x', 24);
7     l_ret := RAWTOHEX (UTL_RAW.cast_to_raw (l_ret));
8     RETURN l_ret;
9* END;
```

В строке 6 генерируется случайная строка печатных символов длиной 24 байта. В строке 7 она приводится к значению типа RAW, а затем преобразуется в шестнадцатеричное значение, как это было сделано в примере для Oracle9*i*. Наконец, строка возвращается для использования в качестве ключа.

Функция работает, но сгенерированная строка недостаточно произвольна с криптографической точки зрения. Однако с учетом того, что ранние версии Oracle не предоставляют средства формирования ключей, единственной альтернативой нашей функции является задание ключа вручную, что невозможно в реальной жизни. Так что предложенный подход является единственно возможным, но пользоваться им следует с осторожностью.

Результат работы функции:

```
Key = 3DA5335923D784F21B0C27B61496D1AD Encrypted Value =
076A5703A745D03934B56F7500C1DCB4
```

Теперь расшифруем зашифрованное значение при помощи того же ключа.

```
DECLARE
   l_key   VARCHAR2 (80)   := '3DA5335923D784F21B0C27B61496D1AD';
   l_enc   VARCHAR2 (2000) := '076A5703A745D03934B56F7500C1DCB4';
   l_dec   VARCHAR2 (2000);
BEGIN
   l_dec := get_dec_val (l_enc, l_key);
   DBMS_OUTPUT.put_line ('Decrypted Value = ' || l_dec);
END;
/
```

Получим такой результат:

```
Decrypted Value = Input Value
```

Используя функции get_key, get_enc_val и get_dec_val мы можем создать полную систему шифрования, о чем будет рассказано в следующем разделе.

Шифрование на практике

Как нам использовать только что полученные знания о шифровании в реальной жизни? Давайте рассмотрим таблицу ACCOUNTS.

```
SQL> DESC accounts
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------------
 ACCOUNT_NO                                NOT NULL NUMBER
 BALANCE                                            NUMBER
 ACCOUNT_NAME                                       VARCHAR2(200)
```

Я хочу защитить данные, зашифровав столбцы BALANCE и ACCOUNT_NAME. Как я уже не раз повторял, наиболее важным элементом шифрования является ключ, и к его выбору следует подходить серьезно. Я могу сгенерировать ключ, использовать его для шифрования значения столбца, затем сохранить где-нибудь ключ и зашифрованное значение для последующего извлечения. Как именно я должен действовать? Есть несколько вариантов.

Можно определить представление для таблицы следующим образом:

1. Добавить в таблицу столбцы ENC_BALANCE и ENC_ACCOUNT_NAME для хранения зашифрованных значений соответствующих столбцов.
2. Добавить еще один столбец — ENC_KEY, для хранения ключа, использованного для шифрования.
3. Создать представление VW_ACCOUNTS:

```
CREATE OR REPLACE VIEW vw_accounts
AS
   SELECT account_no
        , enc_balance AS balance
        , enc_account_name AS account_name
     FROM accounts
/
```

4. Создать триггеры INSTEAD OF для обработки (при необходимости) операций обновления и вставки данных в таблицу.
5. Создать публичный синоним ACCOUNTS для представления VW_ACCOUNTS.
6. Выдать все привилегии на VW_ACCOUNTS и отозвать все привилегии на ACCOUNTS.

В результате подобных манипуляций владелец схемы, а также любые пользователи, имеющие прямые привилегии на таблицу ACCOUNTS, бу-

дут видеть незашифрованные значения. Всем же остальным пользователям будут доступны только зашифрованные значения.

Можно зашифровать сами столбцы и использовать представление для отображения расшифрованных данных, то есть поступить следующим образом:

1. Добавить столбец ENC_KEY для хранения ключа для данной строки.
2. Сохранить в столбцах BALANCE и ACCOUNT_NAME соответствующие зашифрованные значения.
3. Создать представление VW_ACCOUNTS:

```
CREATE OR REPLACE VIEW vw_accounts
AS
   SELECT account_no,
          get_dec_val (balance, enc_key) AS balance,
          get_dec_val (enc_account_name, enc_key) AS account_name
     FROM accounts
/
```

4. Теперь таблица будет содержать зашифрованные значения, а представление – незашифрованные, привилегии на которые можно выдать соответствующим пользователям.
5. Создать триггеры на таблицу для преобразования открытых значений в зашифрованные перед их вставкой или обновлением.

Преимуществом этого подхода является отсутствие необходимости в изменении самой таблицы.

Можно хранить ключи отдельно от таблицы. Оба описанных выше подхода имеют серьезный недостаток – ключ хранится в таблице. И любой, кто имеет доступ на выборку данных из таблицы, сможет увидеть ключ и расшифровать значения. Разумнее хранить ключи вне исходной таблицы, поступая следующим образом:

1. Создать таблицу ACCOUNT_KEYS, содержащую всего два столбца:

   ACCOUNT_NO, который соответствует ACCOUNT_NO записи таблицы ACCOUNTS.

   ENC_KEY – ключ, используемый для шифрования значения.
2. Сделать так, чтобы в реальной таблице ACCOUNTS содержались не открытые, а зашифрованные значения.
3. Создать триггеры для таблицы ACCOUNTS. Триггер AFTER INSERT генерирует ключ, использует его для шифрования реального значения, предоставленного пользователем, заменяет реальное значение на зашифрованное перед сохранением и затем сохраняет ключ в таблице ACCOUNT_KEYS.
4. Создать представление для отображения расшифрованных данных, соединив две таблицы.

Хранение ключей

Хранение ключей – это наиболее ответственная часть шифрования. И если не организовать его должным образом, то весь смысл шифрования как защиты данных может быть утерян. Существует множество вариантов хранения:

Таблицы базы данных

Этот подход, использованный в только что рассмотренном примере, наиболее удобен для работы с ключами. Однако у него есть серьезный недостаток: не существует способа защиты от администратора базы данных, который имеет права доступа к любой таблице.

Файл операционной системы

Файл может создаваться клиентским процессом во время исполнения с помощью встроенного пакета `UTL_FILE` или внешних таблиц, а затем – использоваться для расшифровки. После считывания файл может быть уничтожен. Этот способ обеспечивает более полную защиту от всех пользователей, включая и администратора базы данных.

Пользовательский ввод

Пользователь предоставляет функции ключ для дешифрования во время исполнения. Это наиболее надежный, но и наименее практичный способ из всех трех. Его недостаток в том, что пользователь может забыть ключ, в результате чего зашифрованные данные не удастся расшифровать никогда.

Шифрование в Oracle 10g

Начиная с версии Oracle 10*g* Release 1 Oracle предлагает для поддержки шифрования пакет `DBMS_CRYPTO`. В этом разделе будет рассказано о генерировании ключей и шифровании данных средствами нового пакета. Однако сначала поговорим об отличиях `DBMS_CRYPTO` и `DBMS_OBFUSCATION_TOOLKIT`.

Пакет `DBMS_OBFUSCATION_TOOLKIT` также доступен в Oracle 10*g*, но корпорация Oracle рекомендует пользоваться новым пакетом, так как возможности старого на его фоне оказываются ограниченными.

Различия между DBMS_CRYPTO и DBMS_OBFUSCATION_TOOLKIT

Существует ряд важных отличий `DBMS_CRYPTO` от `DBMS_OBFUSCATION_TOOLKIT`:

Стандарт AES

Алгоритмы DES и DES3 уже устаревают, и многие организации уже перешли на более надежный алгоритм симметричного шифрования – Advanced Encryption Standard (AES). Пакет `DBMS_OBFUSCATION_TOOLKIT`

не поддерживает шифрование по этому новому стандарту, а DBMS_CRYPTO поддерживает.

Потоковое шифрование (Stream cyphering)

Шифрование может применяться к данным поблочно, и такой процесс называется *блочным шифрованием*. Это широко распространенный и простой в использовании метод. Однако не все системы имеют возможность передавать данные равномерными порциями. В качестве примера можно привести зашифрованный контент, передаваемый в широковещательных и других сетях. В таких случаях содержимое должно шифроваться по мере поступления. Это так называемое *потоковое шифрование*. Пакет DBMS_OBFUSCATION_TOOLKIT не поддерживает потоковое шифрование, а DBMS_CRYPTO поддерживает.

Алгоритм защищенного хеширования (Secure Hash Algorithm)

В пакете DBMS_OBFUSCATION_TOOLKIT криптографическое хеширование обеспечивается только функцией Message Digest (MD5), а не современными и надежными алгоритмами, такими как Secure Hash Algorithm 1 (SHA-1), поддерживаемый пакетом DBMS_CRYPTO.

Код аутентификации сообщения (Message Authentication Code)

Использование кода аутентификации сообщений (Message Authentication Code – MAC) позволяет создать хешированное значение для отправляемого сообщения. Впоследствии это значение можно сравнить со значением, вычисленным для сообщения при его получении, чтобы убедиться в целостности сообщения. Это процесс во многом похож на хеширование и отличается от него лишь тем, что для создания хеш-значения необходим ключ (как при шифровании). Пакет DBMS_OBFUSCATION_TOOLKIT не поддерживает создание MAC, а DBMS_CRYPTO поддерживает.

Большие объекты

Пакет DBMS_OBFUSCATION_TOOLKIT не поддерживает большие объекты (LOB) в их *родном* формате, а DBMS_CRYPTO поддерживает. Для шифрования LOB при работе со старым пакетом необходимо сначала преобразовать большие объекты к типу RAW, используя встроенный пакет UTL_RAW. Такой подход усложняет создание приложений.

В некоторых случаях вам придется использовать пакет DBMS_OBFUSCATION_TOOLKIT даже при работе с Oracle 10*g*. Например, если приложение планируется использовать и в Oracle9*i*, и в Oracle 10*g*, то нужно будет пользоваться старым пакетом, так как его поддерживают обе версии. Аналогично, если вы зашифровываете в Oracle 10*g* данные, которые предполагается расшифровывать в Oracle9*i*, придется пользоваться DBMS_OBFUSCATION_TOOLKIT.

Генерирование ключей

Я уже говорил о том, что функция DES3GETKEY пакета DBMS_OBFUSCATION_TOOLKIT, используемая для генерирования ключа шифрования, не дос-

тупна в пакете DBMS_CRYPTO. Вместо нее появляется функция RANDOMBYTES. Так что если вы захотите использовать нашу функцию get_key в Oracle 10*g*, вам придется изменить ее так, чтобы она работала с RANDOMBYTES.

При переходе от одного метода генерирования ключей к другому помните о следующих моментах:

- Входящая в пакет DBMS_OBFUSCATION_TOOLKIT функция DES3GETKEY может сгенерировать ключ, относящийся к типу данных VARCHAR2 или RAW. В пакете DBMS_CRYPTO все шифрование, относящееся к VARCHAR2, реализуется через RAW, поэтому ключ типа VARCHAR2 бесполезен; функция RANDOMBYTES возвращает только ключи типа RAW.
- При работе с пакетом DBMS_CRYPTO нет необходимости в задании начального значения последовательности случайных чисел, как мы делали это в DBMS_OBFUSCATION_TOOLKIT. Функция получает такое начальное назначение из параметра SQLNET.CRYPTO_SEED файла SQLNET.ORA. Соответственно, этот параметр должен иметь действительное значение, являющееся комбинацией символов длиной от 10 до 70 байт, например:

```
    SQLNET.CRYPTO_SEED =
weipcfwe0cu0we98c0wedcpoweqdufd2d2df2dk2d2d23fv43098fpiwef02uc2ecw1x982jd23d908d
```

Давайте посмотрим, как следует изменить функцию get_key так, чтобы она соответствовала новым условиям.

```
/* Файл на веб-сайте: get_key_2.sql */
CREATE OR REPLACE FUNCTION get_key (p_length IN PLS_INTEGER)
   RETURN RAW
IS
   l_ret   RAW (4000);
BEGIN
   l_ret := dbms_crypto.randombytes (p_length);
   RETURN l_ret;
END;
/
```

Обратите внимание на отсутствие параметра *which*. Кроме того, я указал длину генерируемого ключа, что важно для шифрования.

Может оказаться, что по умолчанию на пакет DBMS_CRYPTO не выданы права PUBLIC или для него не существует публичного синонима. Если вы хотите, чтобы все разработчики могли пользоваться пакетом DBMS_CRYPTO, проверьте наличие публичного синонима и соответствующих привилегий. Например, выполните от имени SYS такие операторы:

```
GRANT EXECUTE ON dbms_crypto TO PUBLIC;
CREATE PUBLIC SYNONYM dbms_crypto FOR sys.dbms_crypto;
```

Имейте в виду, что если синоним уже существует, то исполнение оператора завершится с ошибкой, но никаких проблем для базы данных это не создаст.

Функция RANDOMBYTES чрезвычайно проста, так что вы можете посчитать, что нет необходимости в еще большем ее упрощении за счет создания функции-оболочки. Однако существует ряд причин, по которым вы все же можете захотеть поместить RANDOMBYTES внутрь нашей функции get_key:

- Если существующий код уже использует функцию get_key, то необходимо обеспечить обратную совместимость.
- Для того чтобы напечатать название «get_key», необходимо меньше символов, что повышает читабельность кода.
- Единообразие обычно способствует повышению качества кода, одной этой причины достаточно для того, чтобы использовать функцию-оболочку.

В дополнение к генерированию ключей в формате RAW (посредством функции RANDOMBYTES) пакет DBMS_CRYPTO обеспечивает формирование числовых значений, а также двоичных целых. Функция RANDOMINTEGER генерирует двоичный целый ключ, например:

```
l_ret := DBMS_CRYPTO.randominteger;
```

Функция RANDOMNUMBER генерирует ключ целочисленного типа длиной 2^{128}:

```
l_ret := DBMS_CRYPTO.randomnumber;
```

У вас могут возникнуть сомнения в необходимости целого и двоичного целого ключей с учетом того, что шифрование базируется исключительно на типе данных RAW. Они действительно не нужны для шифрования, но могут оказаться полезны для генерирования псевдослучайных чисел в рамках других операций.

Зашифровывание данных

После того как ключ готов, приступаем собственно к шифрованию данных. Воспользуемся для этого программой ENCRYPT пакета DBMS_CRYPTO. Как и ее родственница из пакета DBMS_OBFUSCATION_TOOLKIT, ENCRYPT перегружена и существует как в виде функции, так и процедуры. В отличие от DBMS_OBFUSCATION_TOOLKIT, такая перегрузка обоснована: функция принимает в качестве входного значения только тип данных RAW, в то время как процедура принимает на вход только значения типов CLOB и BLOB.

Давайте рассмотрим простейший пример шифрования значения типа RAW при помощи функции ENCRYPT:

```
DBMS_CRYPTO.encrypt(
   src IN RAW,
   typ IN PLS_INTEGER,
   key IN RAW,
   iv  IN RAW          DEFAULT NULL)
 RETURN RAW;
```

Большинство параметров нам уже знакомо:

`src`

Входное значение, которое подлежит шифрованию.

`key`

Ключ шифрования.

`iv`

Вектор инициализации

Второй параметр, `typ`, является новым и требует более подробного рассмотрения.

Указание типа шифрования

Пакеты `DBMS_OBFUSCATION_TOOLKIT` и `DBMS_CRYPTO` отличаются способами поддержки различных типов шифрования. Пакет `DBMS_OBFUSCATION_TOOLKIT` предлагает специальные функции (и соответствующие процедуры) для каждого алгоритма, например `DESENCRYPT` для DES и `DES3ENCRYPT` для Triple DES. Пакет `DBMS_CRYPTO` предоставляет всего одну функцию, а тип шифрования указывается в параметре. Поддерживаемые алгоритмы шифрования и соответствующие им константы приведены в табл. 4.3. Необходимая константа задается в формате *имя_пакета.имя_константы*. Например, чтобы выбрать алгоритм Triple DES, следует использовать константу `DBMS_CRYPTO.ENCRYPT_3DES`. Обратите внимание, что старый пакет не поддерживает варианты AES и RC4.

Таблица 4.3. Типы шифрования в пакете DBMS_CRYPTO

Константа	Описание	Реальная длина ключа
`ENCRYPT_DES`	Data Encryption Standard (DES)	56
`ENCRYPT_3DES_2KEY`	Modified Triple Data Encryption Standard (3DES); обрабатывает каждый блок трижды, используя 2 ключа	112
`ENCRYPT_3DES`	Triple Data Encryption Standard (3DES); обрабатывает каждый блок трижды	156
`ENCRYPT_AES128`	Advanced Encryption Standard	128
`ENCRYPT_AES192`	Advanced Encryption Standard	192
`ENCRYPT_AES256`	Advanced Encryption Standard	256
`ENCRYPT_RC4`	Потоковое шифрование (единственное)	

Выберите нужный тип шифрования, указав соответствующее значение параметра `typ`. Имейте в виду, что на самом деле это только *часть* значения параметра, который передает также и другую информацию, о чем будет рассказано в следующем разделе.

Типы сцепления

При шифровании данных каждый шифруемый блок может быть зашифрован независимо от остальных или может быть сцеплен с другими для создания более надежной (с криптографической точки зрения) системы. В последнем случае зашифрованное значение лучше защищено. Чтобы выбрать интересующий вас метод сцепления, укажите соответствующую константу из табл. 4.4 в значении параметра `typ`, например `DBMS_CRYPTO.CHAIN_OFB`.

Таблица 4.4. Типы сцепления DBMS_CRYPTO

Константа	Описание
`CHAIN_CBC`	Сцепление блоков шифротекста – Cipher Block Chaining.
`CHAIN_ECB`	Электронная книга кодов – Electronic Code Book.
`CHAIN_CFB`	Шифрование с обратной связью от шифротекста – Cipher Feedback.
`CHAIN_OFB`	Шифрование с обратной связью по выходу – Output Feedback.

Типы дополнения

Как вы помните, при блочном шифровании данные шифруются блоками. Как быть, если длина входных данных не кратна размеру блока? При использовании пакета `DBMS_OBFUSCATION_TOOLKIT` необходимо явно дополнить данные так, чтобы их длина была кратна размеру блока. Однако этот подход не является криптографически надежным. `DBMS_CRYPTO` позволяет указать необходимый тип дополнения. Большинство компаний использует метод PKCS#5.

Чтобы выбрать интересующий вас метод сцепления, укажите соответствующую константу из табл. 4.5 в значении параметра `typ`, `DBMS_CRYPTO.PAD_PKCS5`.

Таблица 4.5. Типы дополнения DBMS_CRYPTO

Константа	Описание
`PAD_PKCS5`	Дополнение средствами криптографической системы с общим ключом (Public Key Cryptography System #5).
`PAD_ZERO`	Дополнение нулями.
`PAD_NONE`	Отсутствие дополнения. Используется при уверенности в том, что длина данных уже кратна размеру шифруемого блока (кратно 8).

Объединение опций в параметре typ

Теперь давайте посмотрим, как свести все эти разнообразные опции воедино. Предположим, что вы выбрали следующие опции шифрования:

Метод дополнения

Дополнение нулями (`PAD_ZERO`).

Алгоритм шифрования

128-битный ключ, алгоритм Advanced Encryption Standard (ENCRYPT_AES128).

Метод сцепления

Блочное шифрование с обратной связью от шифротекста Cipher Feedback (CHAIN_CFB).

Объединяем эти опции в значении параметра typ следующим образом (получается достаточно длинная строка):

```
typ => DBMS_CRYPTO.pad_zero + DBMS_CRYPTO.encrypt_aes128
                            + DBMS_CRYPTO.chain_cfb
```

Аналогично можно задавать любую комбинацию опций функции ENCRYPT. Рассмотрим пример полного вызова функции:

```
DECLARE
   l_enc   RAW(2000);
   l_in    RAW(2000);
   l_key   RAW(2000);
BEGIN
   l_enc :=
      DBMS_CRYPTO.encrypt (src      => l_in,
                           KEY      => l_key,
                           typ      =>   DBMS_CRYPTO.pad_zero
                                       + DBMS_CRYPTO.encrypt_aes128
                                       + DBMS_CRYPTO.chain_cfb
                          );
END;
```

Для удобства работы пакет предлагает две константы с предопределенными комбинациями значений для опций шифрования, сцепления и дополнения (табл. 4.6).

Таблица 4.6. Константы DBMS_CRYPTO с предопределенными наборами значений для параметра typ

Константа	Шифрование	Дополнение	Сцепление блоков
DES_CBC_PKCS5	ENCRYPT_DES	PAD_PKCS5	CHAIN_CBC
DES3_CBC_PKCS5	ENCRYPT_3DES	PAD_PKCS5	CHAIN_CBC

Если надо использовать алгоритм шифрования DES, дополнение по системе PKCS#5 и сцепление блоков шифротекста (CBC), то следует задать константы следующим образом:

```
DECLARE
   l_enc   RAW(2000);
   l_in    RAW(2000);
   l_key   RAW(2000);
BEGIN
   l_enc :=
```

```
      DBMS_CRYPTO.encrypt (src      => l_in,
                           KEY      => l_key,
                           typ      => DBMS_CRYPTO.des_cbc_pkcs5
                          );
END;
/
```

Перепишем исходную функцию шифрования значения.

```
CREATE OR REPLACE FUNCTION get_enc_val (
   p_in_val   IN   RAW,
   p_key      IN   RAW,
   p_iv       IN   RAW := NULL
)
   RETURN RAW
IS
   l_enc_val   RAW (4000);
BEGIN
   l_enc_val :=
      DBMS_CRYPTO.encrypt (src      => p_in_val,
                           KEY      => p_key,
                           iv       => p_iv,
                           typ      =>   DBMS_CRYPTO.encrypt_aes128
                                       + DBMS_CRYPTO.chain_cbc
                                       + DBMS_CRYPTO.pad_pkcs5
                          );
   RETURN l_enc_val;
END;
/
```

Обработка и преобразование данных типа RAW

Приведенная выше функция принимает входные значения типа RAW и предполагает, что вы хотите использовать алгоритм шифрования AES со 128-битным ключом, метод дополнения PKCS#5 и сцепление блоков шифротекста. В реальных приложениях такие ограничения могут оказаться не очень удобными. Например, входные значения обычно имеют формат VARCHAR2 или какой-то числовой формат, а совсем не RAW. Давайте сделаем функцию более общей, обеспечив прием входных значений в формате VARCHAR2 вместо RAW. Функция ENCRYPT принимает входные значения в формате RAW, так что исходные данные придется преобразовывать к типу RAW. Сделаем следующее:

```
l_in := UTL_I18N.string_to_raw (p_in_val, 'AL32UTF8');
```

Может быть, вы помните, что ранее в главе я использовал встроенный пакет UTL_RAW для преобразования значений типа VARCHAR в RAW. В данном случае я использую для такого преобразования функцию UTL_I18N.STRING_TO_RAW, а не UTL_RAW.CAST_TO_RAW. Почему?

Функция ENCRYPT принимает на вход значения типа RAW и, кроме того, требует использования специального набора символов AL32UTF8, ко-

торый не обязательно является набором символов базы данных. Так что фактически необходимо выполнить два преобразования:

- Из текущего набора символов базы данных в набор символов AL32UTF8
- Из VARCHAR2 в RAW

Функция CAST_TO_RAW не умеет выполнять преобразование набора символов, а функция STRING_TO_RAW встроенного пакета UTL_i18n может выполнить оба преобразования.

Пакет UTL_i18n поставляется в составе Oracle Globalization Support и используется для обеспечения глобализации (или интернационализации – internationalization, это англ. слово обычно сокращают как «i18n»: первая буква «i», последняя буква «n» и 18 букв между ними). Подробно о PL/SQL и интернационализации рассказано в главе 24 четвертого издания «Oracle PL/SQL Programming» (Программирование на Oracle PL/SQL).

Функция ENCRYPT возвращает значение типа RAW, которое неудобно хранить в базе данных и обрабатывать. Преобразуем его из RAW в VARCHAR2:

```
l_enc_val := rawtohex(l_enc_val);
```

Выбор алгоритма шифрования

Как вы помните, выбор алгоритма шифрования определяется рядом факторов, например переходом от Oracle9*i* к Oracle 10*g*. Если источником или местом назначения зашифрованных данных является Oracle9*i*, у вас не будет доступа к пакету DBMS_CRYPTO. Придется использовать пакет DBMS_OBFUSCATION_TOOLKIT, который не поддерживает алгоритмы AES. Так что, несмотря на всю надежность и эффективность алгоритмов AES, вам придется воспользоваться чем-то другим, например DES. Для обеспечения дополнительной безопасности можно использовать 3DES (но имейте в виду, что он медленнее DES). Во многих случаях вам придется для удовлетворения разным требованиям выбирать разные алгоритмы шифрования, при том что две другие опции: дополнение и сцепление, будут оставаться неизменными. К сожалению, функция ENCRYPT не позволяет определить тип шифрования напрямую; он должен передаваться в параметре наряду с другими опциями (типами дополнения и сцепления).

Но мы можем сделать это самостоятельно, введя новый параметр (p_algorithm) и включив его в пользовательский пакет шифрования. Этот параметр будет принимать значения только из следующего списка, в котором перечислены алгоритмы шифрования, поддерживаемые DBMS_CRYPTO:

```
DES
3DES_2KEY
3DES
AES128
```

```
AES192
AES256
RC4
```

Переданное значение будет дописано в конец слова «ENCRYPT» и передано функции ENCRYPT:

```
l_enc_algo :=
   CASE p_algorithm
      WHEN 'DES'
         THEN DBMS_CRYPTO.encrypt_des
      WHEN '3DES_2KEY'
         THEN DBMS_CRYPTO.encrypt_3des_2key
      WHEN '3DES'
         THEN DBMS_CRYPTO.encrypt_3des
      WHEN 'AES128'
         THEN DBMS_CRYPTO.encrypt_aes128
      WHEN 'AES192'
         THEN DBMS_CRYPTO.encrypt_aes192
      WHEN 'AES256'
         THEN DBMS_CRYPTO.encrypt_aes256
      WHEN 'RC4'
         THEN DBMS_CRYPTO.encrypt_rc4
   END;
```

Сводим воедино

Объединим все сделанное ранее: теперь функция get_enc_val будет выглядеть следующим образом:

```
/* Файл на веб-сайте: get_enc_val_6.sql */

CREATE OR REPLACE FUNCTION get_enc_val (
   p_in_val      IN   VARCHAR2,
   p_key         IN   VARCHAR2,
   p_algorithm   IN   VARCHAR2 := 'AES128',
   p_iv          IN   VARCHAR2 := NULL
)
   RETURN VARCHAR2
IS
   l_enc_val    RAW (4000);
   l_enc_algo   PLS_INTEGER;
   l_in         RAW (4000);
   l_iv         RAW (4000);
   l_key        RAW (4000);
   l_ret        VARCHAR2 (4000);
BEGIN
   l_enc_algo :=
      CASE p_algorithm
         WHEN 'DES'
            THEN DBMS_CRYPTO.encrypt_des
         WHEN '3DES_2KEY'
            THEN DBMS_CRYPTO.encrypt_3des_2key
         WHEN '3DES'
```

```
            THEN DBMS_CRYPTO.encrypt_3des
         WHEN 'AES128'
            THEN DBMS_CRYPTO.encrypt_aes128
         WHEN 'AES192'
            THEN DBMS_CRYPTO.encrypt_aes192
         WHEN 'AES256'
            THEN dbms_crypto.encrypt_aes256
         WHEN 'RC4'
            THEN DBMS_CRYPTO.encrypt_rc4
      END;
   l_in := utl_i18n.string_to_raw (p_in_val, 'AL32UTF8');
   l_iv := utl_i18n.string_to_raw (p_iv, 'AL32UTF8');
   l_key := utl_i18n.string_to_raw (p_key, 'AL32UTF8');
   l_enc_val :=
      DBMS_CRYPTO.encrypt (src      => l_in,
                           KEY      => l_key,
                           iv       => l_iv,
                           typ      =>   l_enc_algo
                                       + DBMS_CRYPTO.chain_cbc
                                       + DBMS_CRYPTO.pad_pkcs5
                          );
   l_ret := RAWTOHEX (l_enc_val);
   RETURN l_ret;
END;
```

Протестируем созданную функцию.

```
SQL> SELECT get_enc_val ('Test','1234567890123456')
  2>   FROM dual
  3> /

GET_ENC_VAL('TEST','1234567890123456')
--------------------------------------
2137F30B29BE026DFE7D61A194BC34DD
```

Мы создали общую функцию шифрования, которая может (необязательно) принимать на вход алгоритм шифрования и вектор инициализации. Она предполагает использование дополнения по системе PKCS#5 и сцепления по электронной книге кодов, что является общей практикой. Если вам подходят такие характеристики шифрования, то эта программа может стать вашей оболочкой для выполнения рутинных операций шифрования.

Расшифровывание данных

Обратная задача – процесс расшифровывания, при котором зашифрованная строка расшифровывается с помощью того же ключа, который был использован для шифрования. Давайте напишем для расшифровывания новую функцию, `get_dec_val`, используя пакет `DBMS_CRYPTO`.

```
/* Файл на веб-сайте: get_dec_val_2.sql */

CREATE OR REPLACE FUNCTION get_dec_val (
```

```
   p_in_val      IN   VARCHAR2,
   p_key         IN   VARCHAR2,
   p_algorithm   IN   VARCHAR2 := 'AES128',
   p_iv          IN   VARCHAR2 := NULL
)
   RETURN VARCHAR2
IS
   l_dec_val    RAW (4000);
   l_enc_algo   PLS_INTEGER;
   l_in         RAW (4000);
   l_iv         RAW (4000);
   l_key        RAW (4000);
   l_ret        VARCHAR2 (4000);
BEGIN
   l_enc_algo :=
      CASE p_algorithm
         WHEN 'DES'
            THEN DBMS_CRYPTO.encrypt_des
         WHEN '3DES_2KEY'
            THEN DBMS_CRYPTO.encrypt_3des_2key
         WHEN '3DES'
            THEN DBMS_CRYPTO.encrypt_3des
         WHEN 'AES128'
            THEN DBMS_CRYPTO.encrypt_aes128
         WHEN 'AES192'
            THEN DBMS_CRYPTO.encrypt_aes192
         WHEN 'AES256'
            THEN DBMS_CRYPTO.encrypt_aes256
         WHEN 'RC4'
            THEN DBMS_CRYPTO.encrypt_rc4
      END;
   l_in := hextoraw(p_in_val);
   l_iv := utl_i18n.string_to_raw (p_iv, 'AL32UTF8');
   l_key := utl_i18n.string_to_raw (p_key, 'AL32UTF8');
   l_dec_val :=
      DBMS_CRYPTO.decrypt (src      => l_in,
                           KEY      => l_key,
                           iv       => l_iv,
                           typ      =>   l_enc_algo
                                       + DBMS_CRYPTO.chain_cbc
                                       + DBMS_CRYPTO.pad_pkcs5
                          );
   l_ret := utl_i18n.raw_to_char (l_dec_val, 'AL32UTF8');
   RETURN l_ret;
END;
```

Протестируем функцию, попытавшись расшифровать ранее зашифрованное значение:

```
SQL> SELECT get_dec_val ('2137F30B29BE026DFE7D61A194BC34DD',
                         '1234567890123456')
  2>  FROM DUAL
  3>  /
```

```
GET_DEC_VAL('2137F30B29BE026DFE7D61A194BC34DD','1234567890123456')
--------------------------------------------------------------------------------
Test
```

Все отлично, получено исходное значение. Обратите внимание, что использован тот же ключ, что и при зашифровывании. При расшифровывании зашифрованного значения необходимо использовать те же ключ, алгоритм шифрования, тип дополнения и сцепления, что и при зашифровывании.

Вы можете использовать `get_dec_val` как общую программу для расшифровывания зашифрованных значений. Для простоты, удобства управления и обеспечения безопасности я рекомендую поместить этот набор функций шифрования в специальный (собственноручно созданный) пакет.

Прежде чем завершать данный раздел, я хотел бы обратить ваше внимание на важный момент. В двух предыдущих примерах я использовал входные и выходные значения типа `VARCHAR2`. Однако не забывайте о том, что внутри базы данных шифрование выполняется над значениями типа `RAW`, так что нам следует преобразовать данные и ключ из `RAW` в `VARCHAR2`, а затем обратно в `RAW`. Отсутствие таких преобразований упростило наши примеры, но в некоторых случаях оно может оказаться недопустимым (см. примечание «Когда следует использовать шифрование в формате `RAW`» ранее в главе).

Управление ключами в Oracle 10g

Мы изучили основы зашифровывания и расшифровывания, а также способы генерирования ключей. Это была простая часть задачи, ведь по большей части мы использовали имеющиеся программы Oracle и создавали для них оболочки. Теперь пришло время самого серьезного этапа шифрования – управления ключами. Нашим приложениям необходимо обращаться к ключу для расшифровывания зашифрованного значения, и наша задача в том, чтобы сделать механизм доступа как можно более простым. С другой стороны, ключ не должен быть простым настолько, чтобы быть легко разгаданным хакерами. В хорошей системе управления ключами простота доступа к ключу уравновешена мерами предотвращения неавторизованного доступа к нему.

Существует три основных подхода к управлению ключами:

- Использование одного и того же ключа для всей базы данных
- Использование различных ключей для разных строк таблиц с зашифрованными данными
- Комбинированный подход

В последующих разделах будут рассмотрены эти три способа управления ключами.

В этой главе рассматриваются возможности версии Oracle 10*g*, но общая идея также применима и к базе данных Oracle9*i*, так что информация об управлении ключами окажется полезной и для тех, кто пользуется старой версией.

Использование одного ключа

В этом случае для доступа ко всем данным базы данных используется один и тот же ключ. Программа шифрования считывает всего один ключ (оттуда, где он хранится) и шифрует с его помощью все данные, которые следует защитить (рис. 4.3). Существует несколько возможных мест хранения ключа:

В базе данных

Это самый простой способ. Ключ хранится в реляционной таблице, возможно, в специально созданной для этих целей схеме. Благодаря тому, что ключ находится внутри базы данных, автоматически создается его резервная копия, так что старые значения ключа можно получить ретроспективным запросом (flashback query); кроме того, ключ не может быть украден из операционной системы. Но простота подхода является и его слабым местом: ключ – это просто данные в таблице, так что каждый, кто имеет права на редактирование со-

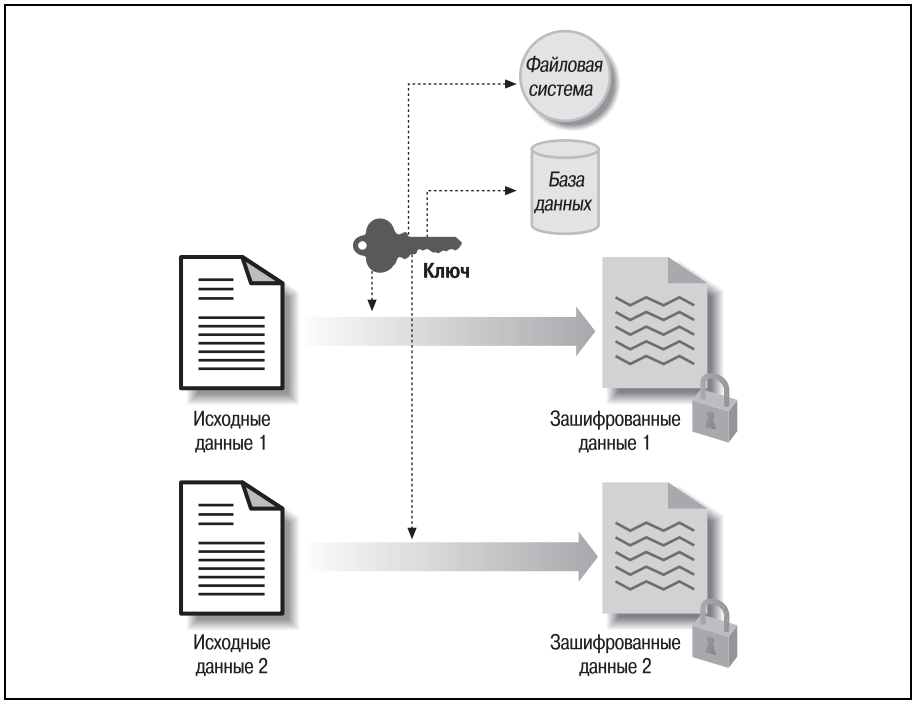


Рис. 4.3. *Использование одного ключа для всей базы данных*

ответствующей таблицы (например, администратор базы данных), может изменить ключ, разрушив тем самым систему шифрования.

В файловой системе

Ключ сохраняется в файле, который затем может быть прочитан процедурой шифрования при помощи встроенного пакета UTL_FILE. Задав соответствующие привилегии для данного файла, вы гарантируете отсутствие возможности его изменения для пользователей базы данных.

На каком-то съемном носителе, подконтрольном конечному пользователю

Это наиболее надежный способ. Никто кроме конечного пользователя не может расшифровать значения или изменить ключ (это относится и к администратору базы данных, и к системному администратору). В качестве примера съемного носителя можно привести USB-карту, DVD-диск или съемный жесткий диск. Основным недостатком съемного носителя является возможность потери или кражи ключа. Ответственность за сохранность ключа целиком ложится на конечного пользователя. Если ключ потерян, то потеряны (безвозвратно) и зашифрованные данные.

Главное преимущество использования единственного ключа заключается в том, что программам шифрования не приходится выбирать ключи из таблиц или сохранять их каждый раз при обработке записи таблицы. Общая производительность увеличивается за счет уменьшения количества циклов процессора и операций ввода-вывода. Главным недостатком этого подхода является его полная зависимость от одного элемента. Если в базу данных проникает злоумышленник и определяет значение ключа, то вся база данных становится уязвимой. Кроме того, если вы захотите сменить ключ, то придется изменить все строки всех таблиц, что может быть достаточно трудоемкой задачей для больших баз данных (рис. 4.3).

Названные недостатки, особенно последствия потери ключа, делают использование этого подхода чрезвычайно редким. Он может быть полезен лишь в отдельных ситуациях. Например, в системе публикации данных, где ключ используется только при передаче данных, затем уничтожается, и для следующей передачи используется уже новый ключ. Такая система может использоваться агентствами финансовой информации при отправке аналитических данных клиентам или, например, в случае, если одно подразделение компании отправляет конфиденциальные корпоративные данные другому подразделению или головному офису.

Использование ключа для каждой строки

Второй подход заключается в использовании разных ключей для всех строк таблицы (рис. 4.4). Такой подход существенно более надежен,

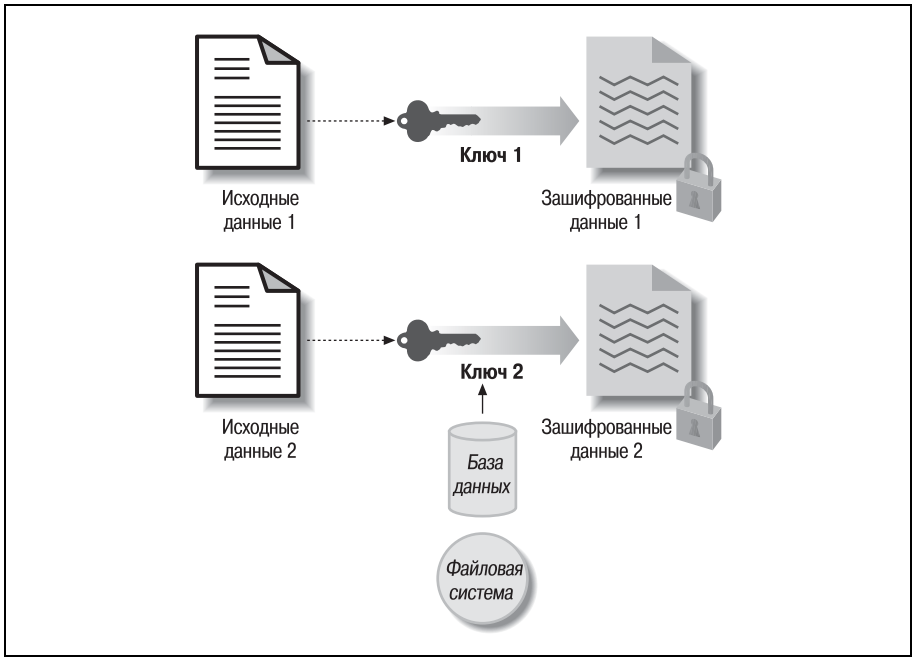


Рис. 4.4. *Использование нового ключа для каждой строки*

чем рассмотренный в предыдущем разделе. Даже если злоумышленнику и удастся похитить ключ, рассекречена будет только одна строка, а не целая таблица или база данных. Однако есть и недостатки: быстрое увеличение количества ключей делает управление ими весьма сложной задачей. Кроме того, страдает производительность, ведь операциям зашифровывания и расшифровывания приходится генерировать или извлекать новый ключ для каждой строки. Тем не менее многие системы шифрования выбирают именно этот подход, поскольку он обеспечивает более высокий уровень безопасности.

Комбинированный подход

В некоторых случаях может не подойти ни один из описанных ранее способов. Давайте выделим положительные и отрицательные стороны каждого из них.

- Использование одного ключа:
  a. Чрезвычайно простое управление ключами. Есть всего один ключ, который нужно создать, прочитать и сделать для него резервную копию.
  b. Ключ можно сохранить во многих местах, к которым удобно обращаться приложениям.

c. С другой стороны, как только ключ украден, вся база данных становится уязвимой.

- Использование отдельного ключа для каждой строки:

a. Количество ключей равно количеству строк, что усложняет управление ключами: больший объем данных нужно хранить, резервировать и т. д.

b. С другой стороны, кража одного ключа приводит к рассекречиванию только одной строки, но не всей базы данных. Общая безопасность системы повышается.

Очевидно, что оба подхода несовершенны. Пользователю нужно найти какое-то компромиссное решение, то есть использовать некий подход, сочетающий в себе качества уже рассмотренных. Можно использовать новый ключ для каждого столбца (при этом к каждой строке будет применен один и тот же ключ), или новый ключ для каждой таблицы независимо от количества столбцов, или новый ключ для каждой схемы, или что-то еще. Количество ключей, используемых любым из предложенных подходов, значительно уменьшится, упростив управление ключами, но возрастет уязвимость данных.

Давайте рассмотрим еще один подход. Будем использовать комбинацию ключей (рис. 4.5):

- Один ключ для каждой строки плюс
- Мастер-ключ для всей базы данных.

Речь не идет о шифровании зашифрованного значения (фактически это невозможно). Несмотря на то что определяется ключ для каждой

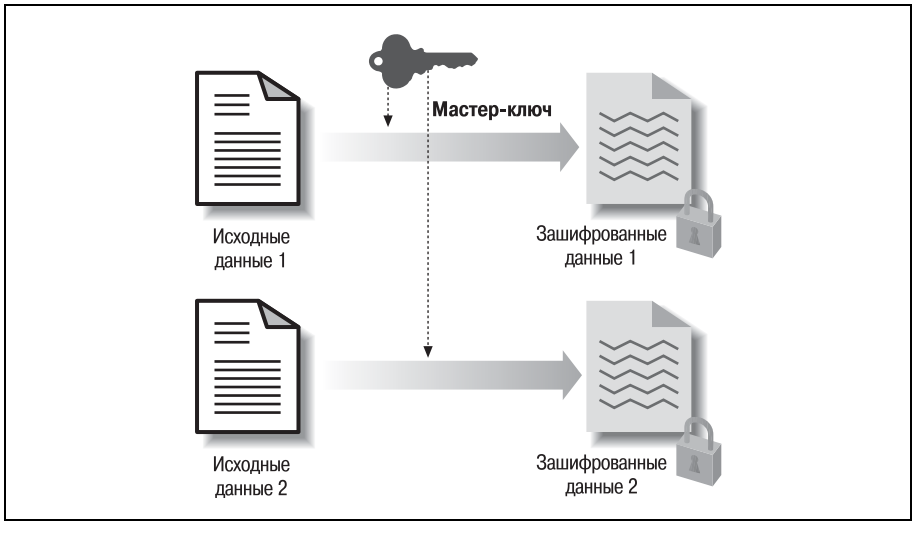


Рис. 4.5. *Использование мастер-ключа*

строки, при шифровании используется *не тот* ключ, который был сохранен для данной строки, а результат побитовой операции XOR (исключающее ИЛИ) для *двух* значений: сохраненного ключа и *мастер-ключа*. Мастер-ключ может храниться отдельно от других ключей, как показано на рис. 4.6. Для успешного расшифровывания зашифрованного значения злоумышленнику придется найти оба ключа.

Встроенный пакет UTL_RAW содержит функцию BIT_XOR, которую можно использовать для выполнения побитовой операции «исключающее ИЛИ». Выполним побитовую операцию XOR для двух значений: 12345678 и 87654321.

```
/* Файл на веб-сайте: bit_xor.sql */
 1  DECLARE
 2     l_bitxor_val   RAW (2000);
 3     l_val_1        VARCHAR2 (2000) := '12345678';
 4     l_val_2        VARCHAR2 (2000) := '87654321';
 5  BEGIN
 6     l_bitxor_val :=
 7        UTL_RAW.bit_xor (utl_i18n.string_to_raw (l_val_1, 'AL32UTF8'),
 8                         utl_i18n.string_to_raw (l_val_2, 'AL32UTF8')
 9                        );
10     DBMS_OUTPUT.put_line (    'Raw Val_1:      '
11                            || RAWTOHEX (utl_i18n.string_to_raw (l_val_1,
12                                                                 'AL32UTF8'
13                                                                )
14                                         )
15                           );
16     DBMS_OUTPUT.put_line (    'Raw Val_2:      '
17                            || RAWTOHEX (utl_i18n.string_to_raw (l_val_2,
18                                                                 'AL32UTF8'
```

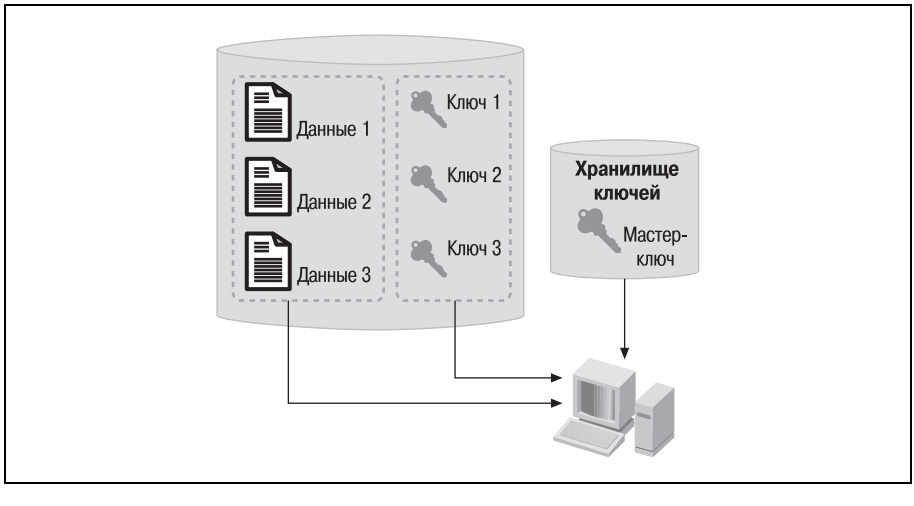


Рис. 4.6. *Хранение мастер-ключа*

```
19                                                              )
20                                          )
21                                );
22     DBMS_OUTPUT.put_line ('After bit XOR: ' || RAWTOHEX (l_bitxor_val));
23  END;
```

Для выполнения побитовой операции сначала преобразуем значения к типу данных RAW (как это сделано в строке 8: вызов функции UTL_I18N.STRING_TO_RAW преобразует значение к типу RAW). В строке 7 вызываем побитовую функцию «исключающее ИЛИ», а в конце выводим два входных значения, преобразованные к типу RAW, а также результат операции XOR.

Результат будет таким:

```
Raw Val_1:     3132333435363738
Raw Val_2:     3837363534333231
After bit XOR: 0905050101050509
```

Обратите внимание, что результат побитовой операции XOR совсем не похож ни на одно из входных значений. Таким образом, на основе двух значений – сохраненного ключа для строки и мастер-ключа – мы можем сгенерировать новый (отличный от них) ключ, который и будет использован для шифрования. Для того чтобы получить этот настоящий ключ (результат операции XOR), вам нужны *оба* значения. Поэтому человек, узнавший одно из значений, не сможет расшифровать значение, полученное с помощью XOR, и соответственно получить реальный ключ.

Данный способ не является повторным шифрованием зашифрованных данных посредством нового ключа. Пакет DBMS_CRYPTO не позволяет повторно зашифровать уже зашифрованное значение. Если бы вы попытались выполнить подобную операцию, то получили бы сообщение об ошибке ORA-2823, информирующее о том, что данные уже зашифрованы.

Изменим нашу программу шифрования так, чтобы в ней использовался мастер-ключ. Добавляем (в строке 6) новую переменную l_master_key, которая принимает значение от пользователя (переменная подстановки &master_key). В строках 15–17 выполняем операцию XOR для ключа и мастер-ключа и используем результат в качестве ключа шифрования вместо переменной l_key в строке 22.

```
/* Файл на веб-сайте: enc_dec_master.sql */
  1  REM
  2  REM Определяем переменную для хранения зашифрованного значения
  3  VARIABLE enc_val varchar2(2000);
  4  DECLARE
  5     l_key          VARCHAR2 (2000) := '1234567890123456';
  6     l_master_key   VARCHAR2 (2000) := '&master_key';
  7     l_in_val       VARCHAR2 (2000) := 'Confidential Data';
```

```
 8     l_mod           NUMBER
 9        :=   DBMS_CRYPTO.encrypt_aes128
10           + DBMS_CRYPTO.chain_cbc
11           + DBMS_CRYPTO.pad_pkcs5;
12     l_enc           RAW (2000);
13     l_enc_key       RAW (2000);
14  BEGIN
15     l_enc_key :=
16        UTL_RAW.bit_xor (utl_i18n.string_to_raw (l_key, 'AL32UTF8'),
17                         utl_i18n.string_to_raw (l_master_key, 'AL32UTF8')
18                        );
19     l_enc :=
20        DBMS_CRYPTO.encrypt (utl_i18n.string_to_raw (l_in_val, 'AL32UTF8'),
21                             l_mod,
22                             l_enc_key
23                            );
24     DBMS_OUTPUT.put_line ('Encrypted=' || l_enc);
25     :enc_val := RAWTOHEX (l_enc);
26  END;
27  /
28  DECLARE
29     l_key           VARCHAR2 (2000) := '1234567890123456';
30     l_master_key    VARCHAR2 (2000) := '&master_key';
31     l_in_val        RAW (2000)      := HEXTORAW (:enc_val);
32     l_mod           NUMBER
33        :=   DBMS_CRYPTO.encrypt_aes128
34           + DBMS_CRYPTO.chain_cbc
35           + DBMS_CRYPTO.pad_pkcs5;
36     l_dec           RAW (2000);
37     l_enc_key       RAW (2000);
38  BEGIN
39     l_enc_key :=
40        UTL_RAW.bit_xor (utl_i18n.string_to_raw (l_key, 'AL32UTF8'),
41                         utl_i18n.string_to_raw (l_master_key, 'AL32UTF8')
42                        );
43     l_dec := DBMS_CRYPTO.decrypt (l_in_val, l_mod, l_enc_key);
44     DBMS_OUTPUT.put_line ('Decrypted=' || utl_i18n.raw_to_char (l_dec));
45  END;
```

Результатом выполнения приведенного фрагмента кода будут такие выходные данные. Обратите внимание, что сначала я задаю мастер-ключ для шифрования значения, а затем тот же самый мастер-ключ для расшифровывания.

```
Enter value for master_key: MasterKey0123456
old   3:     l_master_key varchar2(2000) := '&master_key';
new   3:     l_master_key varchar2(2000) := 'MasterKey0123456';
Encrypted=C2CABD4FD4952BC3ABB23BD50849D0C937D3EE6659D58A32AC69EFFD4E83F79D

PL/SQL procedure successfully completed.

Enter value for master_key: MasterKey0123456
```

```
old   3:     l_master_key varchar2(2000) := '&master_key';
new   3:     l_master_key varchar2(2000) := 'MasterKey0123456';
Decrypted=ConfidentialData

PL/SQL procedure successfully completed.
```

Программа запрашивает мастер-ключ, я предоставляю ей корректное значение и получаю корректное значение. Но что будет, если я укажу недействительный мастер-ключ?

```
Enter value for master_key: MasterKey0123456
old   3:     l_master_key varchar2(2000) := '&master_key';
new   3:     l_master_key varchar2(2000) := 'MasterKey';
Encrypted=C2CABD4FD4952BC3ABB23BD50849D0C937D3EE6659D58A32AC69EFFD4E83F79D

PL/SQL procedure successfully completed.

Enter value for master_key: MasterKey0123455
old   3:     l_master_key varchar2(2000) := '&master_key';
new   3:     l_master_key varchar2(2000) := 'WrongMasterKey';
declare
*
ERROR at line 1:
ORA-28817: PL/SQL function returned an error.
ORA-06512: at "SYS.DBMS_CRYPTO_FFI", line 67
ORA-06512: at "SYS.DBMS_CRYPTO", line 41
ORA-06512: at line 15
```

Мы видим сообщение об ошибке: использование неверного мастер-ключа приводит к тому, что зашифрованные данные не расшифровываются. Наш усовершенствованный механизм базируется на двух разных ключах, и оба ключа необходимы для успешной расшифровки. Если вы спрячете мастер-ключ, то этого будет достаточно для предотвращения неавторизованного расшифровывания.

Т. к. мастер-ключ хранится на клиентском компьютере и пересылается по сети, то потенциальный злоумышленник может использовать специальное средство (снифер) для перехвата значения в момент его передачи. Защититься от этого можно следующими способами:

- Можно создать виртуальную локальную сеть (virtual local area network – VLAN) между сервером приложений и сервером базы данных. VLAN в значительной мере защищает сетевой трафик между серверами.
- Вы можете изменить мастер-ключ каким-то предопределенным способом, например, изменив порядок символов на обратный; тогда злоумышленник, получив ключ, переданный по сети, не получает реально используемый мастер-ключ.
- Наконец, действительно надежным решением является использование опции расширенной безопасности ASO – Oracle Advanced Security Option (предоставляемой за дополнительную плату), обеспечивающей безопасность трафика между клиентом и сервером.

Защита от администратора базы данных?

Следует ли защищать зашифрованные данные от собственного администратора базы данных? Такой вопрос возникает при проектировании системы, и вы должны как-то на него ответить.

Ключ хранится в базе данных или в файловой системе. Если ключ хранится в базе данных, то администратор имеет возможность расшифровать любые зашифрованные данные (так как у него есть права на выборку из *любой* таблицы, в том числе и из таблицы, в которой хранятся ключи). Если ключ хранится в файловой системе, он должен быть доступен владельцу программного обеспечения Oracle, так чтобы его можно было прочитать с помощью пакета `UTL_FILE`, к которому может иметь доступ администратор базы данных. Таким образом, где бы ни хранился ключ, попытка защиты зашифрованных данных от администратора базы данных представляется безуспешной. Оправдан ли подобный риск в вашей компании? Ответ зависит от проводимой политики безопасности и руководящих документов. В большинстве случаев администраторам доверяют, но это может быть предметом обсуждения. В некоторых случаях необходимо защищать зашифрованные данные даже от администраторов баз данных.

В таком случае придется хранить ключи там, куда у администратора базы данных нет доступа, например на сервере приложений. Но такими ключами будет сложно управлять. Необходимо будет обеспечить их резервное копирование и защиту от кражи.

Можно использовать более сложную систему управления, используя мастер-ключ. Мастер-ключ может быть помещен в цифровой «бумажник» (*wallet*), а приложение будет запрашивать ключ каждый раз при шифровании данных. Ключ будет недоступен администратору базы данных, но вся система станет более сложной, и время обработки возрастет.

Если ваша цель состоит в том, чтобы воспрепятствовать администратору базы данных изменить ключ, не запрещая просмотр ключа, можно также использовать механизм с мастер-ключом. Мастер-ключ может быть помещен в файловую систему, которая доступна владельцу программного обеспечения Oracle только для чтения. В этом случае база данных (и администратор) сможет использовать ключ для шифрования, но изменить ключ администратор не сможет.

Для того чтобы обеспечить управляемость системы (особенно если вы хотите быть уверены в том, что влияние на ваши приложения минимально), следует хранить ключи или в файловой системе, или внутри таблицы базы данных. В этом случае скрыть их от администратора базы данных не удастся.

Не существует совершенного механизма управления ключами шифрования. При выборе такого механизма следует руководствоваться спецификой конкретного приложения и пытаться найти компромисс между простотой доступа к ключам и безопасностью данных. Три рассмотренных нами подхода описывают три основных механизма управления ключами, на основе которых вы сможете разработать собственный подход к управлению ключами шифрования. Возможно, у вас появятся более удачные идеи, лучше подходящие к вашей конкретной ситуации. Например, возможно применение гибридного подхода, с использованием различных ключей для наиболее важных таблиц.

Прозрачное шифрование данных в Oracle 10g Release 2

Если вы храните ключ шифрования и зашифрованные данные в базе данных, то возникает еще одна потенциальная угроза безопасности. При краже дисков, на которых хранится вся база данных, все данные сразу же рассекречиваются. Для того чтобы обойти эту проблему, следует хранить ключ отдельно, вне диска, на котором хранятся зашифрованные этим ключом данные.

Если ваша база данных полностью изолирована, вы, возможно, не видите смысла в шифровании данных. Однако вы можете захотеть обезопасить себя от кражи диска. В качестве одного из решений можно предложить создание представления для отображения расшифрованного значения. В этом случае, если ключ хранится отдельно, физическая кража диска не приведет к рассекречиванию данных. Данный подход возможен, но требует тщательной, трудоемкой настройки.

Для разрешения подобных ситуаций в Oracle 10*g* Release 2 введена новая возможность прозрачного шифрования данных (Transparent Data Encryption – TDE). TDE использует сочетание двух ключей: мастер-ключ хранится вне базы данных в «бумажнике», кроме того, есть еще ключ для каждой таблицы. Для всех строк таблицы используется один и тот же ключ, ключ каждой таблицы уникален (рис. 4.7).

TDE подразумевает, что вы задаете подмножество столбцов для шифрования. Например, если в таблице 4 столбца, как показано на рис. 4.7, и шифруются столбцы 2 и 3, то Oracle сгенерирует ключ и использует его для шифрования данных столбцов. На диске столбцы 1 и 4 будут сохранены в открытом виде, а два других столбца – в зашифрованном.

При выборе пользователем зашифрованных столбцов Oracle незаметно извлекает ключ из «бумажника», расшифровывает столбцы и показывает их пользователю. Если диск с данными похищен, их невозможно извлечь без ключей, которые хранятся в «бумажнике», зашифрованном мастер-ключом, который и сам сохранен не в виде открытого текста. В результате вор не сможет расшифровать данные, даже если украдет диски или скопирует файлы.

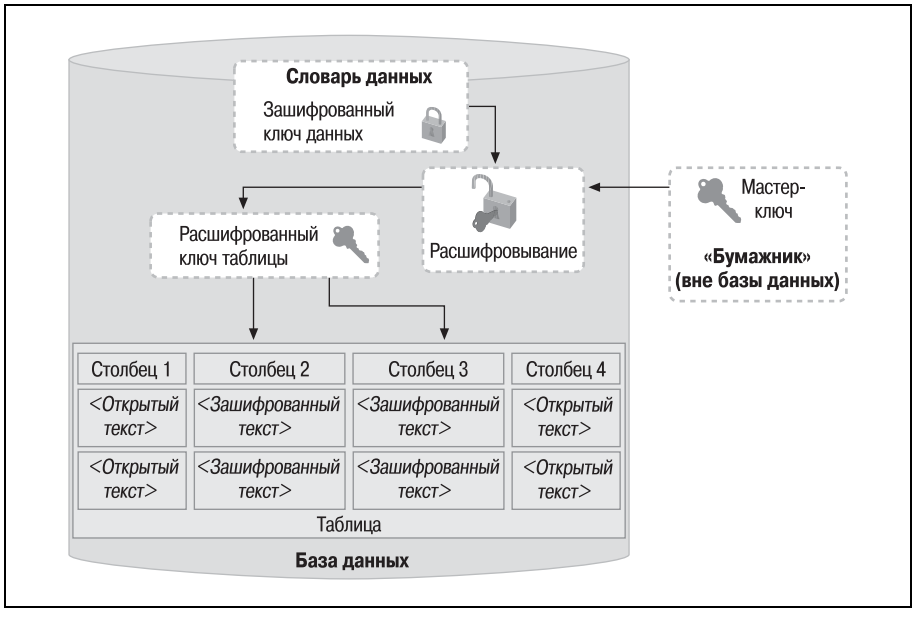


Рис. 4.7. *Модель прозрачного шифрования данных*

Задача Transparent Data Encryption (TDE) состоит в обеспечении защиты данных, хранящихся на таких носителях, как диски и магнитные ленты, которая необходима в соответствии со многими национальными или международными нормативными документами и правилами, такими как Sarbanes-Oxley, HIPAA, Visa Cardholder Information Security Program и т. д.

TDE не является полномасштабной системой шифрования и не должна использоваться в таком качестве. Например, обратите внимание на то, что зашифрованные столбцы расшифровываются в любом случае, вне зависимости от того, кто выбирает данные, что вряд ли удовлетворяет вашим требованиям безопасности. Для получения комплексного решения следует создать собственный инструмент, используя описанные в главе приемы.

Для того чтобы воспользоваться преимуществами TDE, добавьте предложение `ENCRYPT` (доступное только в Oracle 10*g* Release 2) для каждого шифруемого столбца в оператор создания вашей таблицы:

```
/* Файл на веб-сайте: cr_accounts.sql */
CREATE TABLE accounts
(
   acc_no       NUMBER       NOT NULL,
   first_name   VARCHAR2(30) NOT NULL,
   last_name    VARCHAR2(30) NOT NULL,
   SSN          VARCHAR2(9)             ENCRYPT USING 'AES128',
```

```
    acc_type     VARCHAR2(1)  NOT NULL,
    folio_id     NUMBER                  ENCRYPT USING 'AES128',
    sub_acc_type VARCHAR2(30),
    acc_open_dt  DATE         NOT NULL,
    acc_mod_dt   DATE,
    acc_mgr_id   NUMBER
);
```

В данном случае столбцы SSN и FOLIO_ID шифруются при помощи AES-алгоритма со 128-битным ключом. Предложение ENCRYPT USING в определении столбца указывает на необходимость перехвата значений в виде открытого текста, их шифрования и сохранения в зашифрованном виде. Когда пользователь выбирает данные из таблицы, значение неявно (прозрачно) расшифровывается.

Нельзя использовать прозрачное шифрование в таблицах, принадлежащих пользователю SYS.

Настройка TDE

Прежде чем начать использовать TDE, необходимо создать «бумажник», в котором будет храниться мастер-ключ, и обеспечить его безопасность. Рассмотрим процесс управления «бумажниками» пошагово.

1. Выбрать местоположение «бумажника».

   Перед первым применением TDE необходимо создать «бумажник», в котором будет храниться мастер-ключ. По умолчанию «бумажник» создается в каталоге $ORACLE_BASE/admin/$ORACLE_SID/wallet. Вы можете выбрать другой каталог, указав его в файле SQLNET.ORA. Например, если вы хотите, чтобы «бумажник» хранился в каталоге /oracle_wallet, добавьте приведенные ниже строки в файл SQLNET.ORA. В нашем случае я буду считать, что выбран каталог по умолчанию.

   ```
   ENCRYPTION_WALLET_LOCATION =
     (SOURCE=
         (METHOD=file)
         (METHOD_DATA=
            (DIRECTORY=/oracle_wallet)))
   ```

   Убедитесь в том, что «бумажник» включен в процесс резервного копирования.

2. Установить пароль для «бумажника».

   Теперь вам нужно создать «бумажник» и указать пароль для доступа к нему. Все это можно сделать, выполнив один оператор:

   ```
   ALTER SYSTEM SET ENCRYPTION KEY IDENTIFIED BY "pooh";
   ```

   Данный оператор выполняет три действия:

   a. Создает «бумажник» в каталоге, определенном в шаге 1.

b. Устанавливает для «бумажника» пароль – «pooh».

c. Открывает «бумажник» для сохранения и извлечения ключей средствами TDE.

Пароль является регистрозависимым и должен заключаться в двойные кавычки.

3. Открыть «бумажник».

На предыдущем шаге бумажник был открыт для работы с ним. Однако после того как бумажник создан, вам не придется пересоздавать его. После запуска базы данных вам нужно будет только открыть «бумажник», используя установленный пароль:

```
ALTER SYSTEM SET ENCRYPTION WALLET OPEN IDENTIFIED BY "pooh";
```

Вы можете закрыть «бумажник» следующей командой:

```
ALTER SYSTEM SET ENCRYPTION WALLET CLOSE;
```

Открытие «бумажника» необходимо для работы средств TDE. Если «бумажник» не открыт, то все незашифрованные столбцы будут доступны, а зашифрованные столбцы – недоступны.

Использование TDE для уже существующих таблиц

В предыдущем разделе мы видели, как можно использовать TDE при создании новой таблицы. Так же можно зашифровать столбец существующей таблицы. Для шифрования столбца SSN таблицы ACCOUNTS используем такой оператор:

```
ALTER TABLE accounts MODIFY (ssn ENCRYPT);
```

Данный оператор выполняет два действия:

- Создает ключ для столбца SSN.
- Преобразует все значения столбца в зашифрованный формат.

Шифрование выполняется внутри базы данных. По умолчанию используется алгоритм AES со 192-битным ключом. Вы можете выбрать другой алгоритм шифрования, указав его название в операторе. Например, чтобы выбрать алгоритм AES со 128-битным ключом, используйте такой оператор:

```
ALTER TABLE accounts MODIFY (ssn ENCRYPT USING 'AES128');
```

В качестве параметров также можно было бы указать AES128, AES256 или 3DES168 (для трехпроходного механизма DES со 168-битным ключом). Посмотрим на таблицу после шифрования столбца:

```
SQL> DESC accounts
 Name      Null? Type
 --------- ----- ------------
 ACC_NO          NUMBER
```

```
ACC_NAME        VARCHAR2(30)
SSN             VARCHAR2(9) ENCRYPT
```

Обратите внимание на ключевое слово ENCRYPT после типа данных. Для поиска зашифрованных столбцов базы данных используйте новое представление словаря данных DBA_ENCRYPTED_COLUMNS.

А как обстоит дело с производительностью при работе с TDE? При обращении к незашифрованным столбцам никаких дополнительных накладных расходов не возникает. При обращении к зашифрованным столбцам можно ожидать небольшого увеличения накладных расходов. Если потребность в шифровании пропадает, его можно отключить следующим образом:

```
ALTER TABLE accounts MODIFY (ssn DECRYPT);
```

Управление ключами и паролями для TDE

Что будет, если кто-то каким-то образом узнает ваши TDE-ключи? Можно пересоздать зашифрованные значения, выполнив всего один оператор. В этом операторе можно также выбрать другой алгоритм шифрования, например AES256:

```
ALTER TABLE accounts REKEY USING 'aes256';
```

Если кто-то узнает пароль «бумажника», вы можете изменить его, используя программу Oracle Wallet Manager. Введите owm в командной строке, и диспетчер «бумажников» будет запущен (рис. 4.8). Выберите в главном меню Wallet→Open и укажите место хранения «бумажника» и его пароль. Для изменения пароля выберите Wallet→Change Password. Имейте в виду, что изменение пароля не приводит к изменению ключей.

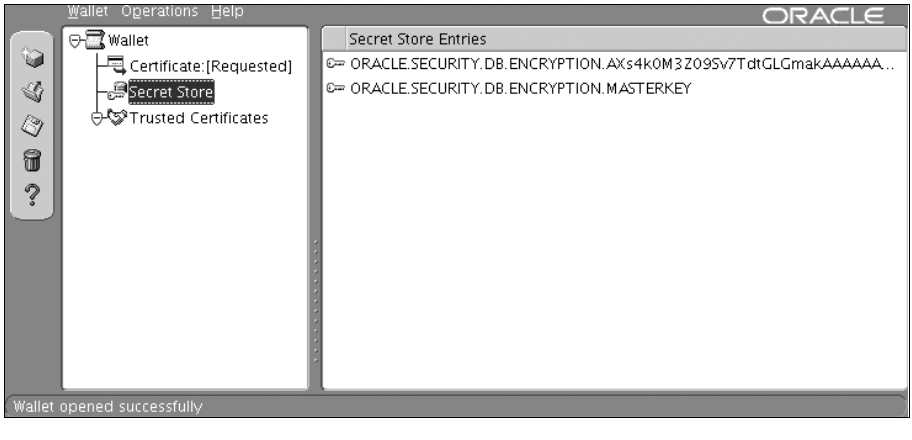


***Рис. 4.8.** Oracle Wallet Manager*

Добавим «соли»

Шифрование предназначено для сокрытия данных, но бывает так, что зашифрованные данные легко угадать из-за повторений в исходных данных. Например, таблица с информацией о зарплатах весьма вероятно содержит повторяющиеся значения. В этом случае зашифрованные значения также будут одинаковыми. Даже если злоумышленник не сможет расшифровать реальные значения, он сможет определить, в каких записях присутствуют одинаковые зарплаты, а эта информация может быть значимой. Для предотвращения таких ситуаций к данным добавляется *соль* (*salt*), благодаря которой зашифрованные значения будут различаться даже для одинаковых исходных данных. TDE использует *соль* по умолчанию.

В некоторых случаях структуры данных могут способствовать повышению производительности базы данных, а добавление «соли» может ее ухудшить. Например, индексы b-tree ускоряют поиск по условию `LIKE`, как в следующем запросе:

```
SELECT ... FROM accounts WHERE ssn LIKE '123%';
```

В данном случае индексам b-tree для получения данных придется пройти только по одной ветви дерева, так как все номера счетов начинаются с цифр `123`. Если «соль» добавлена, то реальные значения будут распределены по всей структуре b-tree, что сделает просмотр индекса более трудоемким, и весьма вероятно, что оптимизатор выберет полный просмотр таблицы. В этом случае придется убрать «соль» из индексированных столбцов, например, так:

```
ALTER TABLE accounts MODIFY (ssn ENCRYPT NO SALT);
```

Удаление «соли» незначительно влияет на безопасность, поэтому, скорее всего, возможная уязвимость не перевешивает того повышения производительности, которое обеспечивается индексированием.

Использование средств TDE невозможно для столбцов, обладающих одним из приведенных ниже свойств:

- Тип данных `BLOB` или `CLOB`.
- Использование в индексах, отличных от обычных индексов b-tree, таких как bitmap-индексы, индексы на основе функций и т. д.
- Использование в ключах секционирования.

Отсутствие возможности использования средств TDE в подобных случаях является еще одной причиной, по которой TDE не может использоваться для всех видов шифрования.

Криптографическое хеширование

Шифрование обеспечивает доступ к вашим данным только авторизованным пользователям. Это достигается за счет маскировки секрет-

ных данных. Однако в некоторых случаях в маскировке нет необходимости, хочется лишь защитить данные от изменения. Предположим, вы сохранили информацию о платежах вашим поставщикам. Сами по себе данные не настолько секретны, чтобы их шифровать, но хочется иметь уверенность в том, что никто не изменит цифры, с тем чтобы увеличить размер платежа. Как это сделать? Ответом является *криптографическое хеширование*. Давайте начнем знакомство с ним с примера из реальной жизни.

Дело о подозрительном сэндвиче

Предположим, что вы оставили свой сэндвич на столе, когда пошли забрать из факсимильного аппарата важный документ. Вернувшись, вы замечаете, что сэндвич несколько сдвинут влево. Кто-то трогал ваш сэндвич, возможно, подложив в него барбитуратов, чтобы вывести вас из игры и завладеть вашей новой чудесной беспроводной мышью? А может быть, он охотился на книгу по PL/SQL, спрятанную в ящике стола? А может быть, в сэндвиче если не наркотики, так песок? В мозгу прокручивается множество вариантов произошедшего, и есть уже не хочется.

Чтобы развеять свои сомнения, вы решаете проверить целостность сэндвича. Вы настолько осторожны и предусмотрительны, что заранее взвесили свой сэндвич и записали его вес с точностью до десятого знака после запятой. Опасаясь возможного изменения сэндвича, вы снова взвешиваете его и сравниваете результаты. Полное совпадение, вплоть до 10 знака после запятой. Какое облегчение! Если бы кто-то действительно что-то сделал с сэндвичем (например, добавил в него песка или барбитуратов), его вес обязательно бы изменился, свидетельствуя о вмешательстве.

Будем аккуратны с терминологией. Вы не «прятали» сэндвич (то есть не *зашифровывали* его); вы просто придумали собственный способ вычисления определенного значения, сопоставленного данному сэндвичу. И сравнили начальное и конечное значения. Исследуемое значение могло быть получено с помощью любого алгоритма; в данном случае вы выбрали взвешивание сэндвича.

Если бы речь шла о данных, а не о бутерброде с мясом, вы также могли бы получить некоторое значение по определенному алгоритму. Такой процесс называется *хешированием*. Отличие хеширования от шифрования в том, что хеширование – это однонаправленный процесс. Вы можете расшифровать зашифрованные данные, но «расхешировать» хеш-значение невозможно. Если вы несколько раз хешируете один и тот же элемент данных, результат будет неизменным при любом количестве выполнений. Если данные каким-то образом меняются, генерируемое хеш-значение изменится, свидетельствуя о «порче» данных.

Теоретически всегда есть риск того, что у двух разных элементов данных окажутся одинаковые хеш-значения, но такую вероятность можно минимизировать, используя достаточно изощренные алгоритмы хеши-

рования. Одним из таких алгоритмов является MD (Message Digest – дайджест сообщений). Одна из разновидностей этого алгоритма – MD5, некоторое время была стандартом, но оказалось, что она не обеспечивает достаточной защищенности данных. Сейчас более распространенным является новый стандарт SHA-1 (алгоритм безопасного хеширования – версия 1, Secure Hash Algorithm Version 1), который доступен в Oracle 10*g*.

Хеширование MD5 в Oracle9i

Давайте посмотрим, как можно использовать хеширование при администрировании реальных баз данных. Отправляя куда-то конфиденциальную информацию, вы можете вычислить хеш-значение до отправки и выслать его тому же адресату в другом отправлении. Получатель может вычислить хеш-значение для полученных данных и сравнить его с отправленным вами хеш-значением.

В Oracle9*i* пакет DBMS_OBFUSCATION_TOOLKIT содержит функцию хеширования MD5, реализующую протокол Message Digest. Для хеширования строки делаем следующее:

```
DECLARE
   l_hash     VARCHAR2 (2000);
   l_in_val   VARCHAR2 (2000);
BEGIN
   l_in_val := 'Account Balance is 12345.67';
   l_hash := DBMS_OBFUSCATION_TOOLKIT.md5 (input_string => l_in_val);
   l_hash := RAWTOHEX (UTL_RAW.cast_to_raw (l_hash));
   DBMS_OUTPUT.put_line ('Hashed Value = ' || l_hash);
END;
/
```

Я передал функции простую строку «Account Balance is 12345.67» и получил ее хеш-значение. Функция MD5 возвращает значение типа VARCHAR2, но (как и при рассмотренном ранее шифровании) это значение содержит управляющие символы. Поэтому необходимо преобразовать его к типу RAW, а затем к шестнадцатеричному формату для удобства хранения. Приведенный выше фрагмент кода возвращает такое значение:

```
Hashed Value = A09308E539C35C97CD612E918BA58B4C
```

Видны два важных отличия хеширования от шифрования:

- При хешировании входную строку не нужно дополнять до определенной длины, как это делается при шифровании.
- При хешировании не используется ключ. Раз ключа нет, его не нужно хранить или вводить, что делает систему хеширования чрезвычайно простой.

Итак, я могу захотеть получить хешированное значение и отправить его некоторому адресату. Можно создать и сохранить функцию, которая будет это делать. Используем тот же пример кода и создадим функцию:

```
/* Файл на веб-сайте: get_hash_val_9i.sql */
CREATE OR REPLACE FUNCTION get_hash_val (p_in VARCHAR2)
   RETURN VARCHAR2
IS
   l_hash   VARCHAR2 (2000);
BEGIN
   l_hash :=
      RAWTOHEX
         (UTL_RAW.cast_to_raw
              (DBMS_OBFUSCATION_TOOLKIT.md5 (input_string      => p_in)
               )
         );
   RETURN l_hash;
END;
```

Давайте получим от функции какие-нибудь типичные данные.

```
BEGIN
   DBMS_OUTPUT.put_line (   'Hashed = '
                         || get_hash_val ('Account Balance is 12345.67')
                        );
   DBMS_OUTPUT.put_line (   'Hashed = '
                         || get_hash_val ('Account Balance is 12345.67')
                        );
END;
```

Результат будет таким:

```
Hashed = A09308E539C35C97CD612E918BA58B4C
Hashed = A09308E539C35C97CD612E918BA58B4C
```

Как видите, для одной и той же строки ввода функция каждый раз возвращает *идентичные* значения: этот факт можно использовать для проверки целостности элемента данных. Обратите внимание, что я говорю о целостности *данных*, а не базы данных. Целостность базы данных обеспечивается механизмом обеспечения ограничений целостности и транзакций Oracle. Законный пользователь может изменить значение таким образом, что имеющиеся ограничения целостности не будут нарушены, но данные (не база данных) при этом могут быть повреждены. Например, если кто-то специальным оператором SQL изменит баланс счета с 12345,67 долларов на 21345,67 долларов, то этот факт может остаться незамеченным до тех пор, пока не будет проведено расследование.

Если хеш-значение для столбца, хранящего номер социального страхования, вычислить заранее и сохранить, а после извлечения данных из столбца повторно вычислить хеш-значение и сравнить с сохраненным, то несовпадение значений будет свидетельствовать о возможных злонамеренных операциях с данными. Давайте посмотрим, как это работает.

```
DECLARE
   l_data   VARCHAR2 (200);
BEGIN
```

```
    l_data := 'Social Security Number = 123-45-6789';
    DBMS_OUTPUT.put_line ('Hashed = ' || get_hash_val (l_data));
    --
    -- кто-то изменил данные
    --
    l_data := 'Social Security Number = 023-45-6789';
    DBMS_OUTPUT.put_line ('Hashed = ' || get_hash_val (l_data));
END;
```

Вывод будет таким:

```
Hashed = 098D833A81B279E54992BFB1ECA6E428
Hashed = 6682A974924B5611FA9D809357ADE508
```

Как видите, хеш-значения отличаются. Результирующее хеш-значение изменится при любом изменении данных, даже если собственно значение не изменится. Хеш-значение изменится при изменении пробела, знака препинания или любого другого элемента.

Теоретически возможно получение одного и того же хеш-значения для двух разных входных значений. Однако, используя такие общераспространенные алгоритмы, как MD5 и SHA-1, вы обеспечиваете чрезвычайно малую вероятность хеш-конфликта – порядка 1 из 10^{38} (в зависимости от выбранного алгоритма). Если вы не можете допустить наличия даже столь малой вероятности, то вам придется реализовать логику разрешения конфликтов для хеш-функций.

Хеширование SHA–1 в Oracle 10g

Я уже упоминал о том, что протокол MD5 считается недостаточно надежным для современной защиты данных и вместо него часто используется SHA-1. Алгоритм SHA-1 не поддерживается в пакете DBMS_OBFUSCATION_TOOLKIT. В версии Oracle 10*g* вы можете использовать функцию HASH пакета DBMS_CRYPTO для выполнения хеширования по алгоритму SHA-1. Рассмотрим объявление функции:

```
DBMS_CRYPTO.hash (
   src in raw,
   typ in pls_integer)
 return raw;
```

Функция HASH принимает на вход только значения типа RAW, поэтому необходимо преобразовать входную символьную строку к типу RAW, что мы и делаем так же, как ранее для шифрования.

```
l_in := utl_i18n.string_to_raw (p_in_val, 'AL32UTF8');
```

Теперь преобразованную строку можно передать в хеш-функцию.

Во втором параметре typ (который должен быть объявлен с типом PLS_INTEGER) вы указываете алгоритм хеширования. Выбрать можно один из алгоритмов, упомянутых в табл. 4.7.

Таблица 4.7. Алгоритмы хеширования в пакете DBMS_CRYPTO

Константа	Описание
DBMS_CRYPTO.HASH_MD5	Message Digest 5
DBMS_CRYPTO.HASH_MD4	Message Digest 4
DBMS_CRYPTO.HASH_SH1	Secure Hashing Algorithm 1

Например, чтобы получить хеш-значение для переменной типа RAW, запишем функцию следующим образом:

```
/* Файл на веб-сайте: get_sha1_hash_val.sql */
CREATE OR REPLACE FUNCTION get_sha1_hash_val (p_in RAW)
   RETURN RAW
IS
   l_hash   RAW (4000);
BEGIN
   l_hash := DBMS_CRYPTO.HASH (src => p_in, typ => DBMS_CRYPTO.hash_sh1);
   RETURN l_hash;
END;
/
```

Для использования алгоритма MD5 следует изменить значение параметра typ с DBMS_CRYPTO.HASH_SH1 на DBMS_CRYPTO.HASH_MD5. Свою функцию получения хеш-значения я могу построить так, чтобы она могла принимать любой алгоритм хеширования.

Наконец, возвращаемое значение типа RAW необходимо преобразовать к типу VARCHAR2.

```
l_enc_val := rawtohex (l_enc_val, 'AL32UTF8');
```

Соединяем фрагменты кода и получаем такую функцию:

```
/* Файл на веб-сайте: get_hash_val_10g.sql */
CREATE OR REPLACE FUNCTION get_hash_val (
   p_in_val      IN   VARCHAR2,
   p_algorithm   IN   VARCHAR2 := 'SH1'
)
   RETURN VARCHAR2
IS
   l_hash_val    RAW (4000);
   l_hash_algo   PLS_INTEGER;
   l_in          RAW (4000);
   l_ret         VARCHAR2 (4000);
BEGIN
   l_hash_algo :=
      CASE p_algorithm
         WHEN 'SH1'
            THEN DBMS_CRYPTO.hash_sh1
         WHEN 'MD4'
            THEN DBMS_CRYPTO.hash_md4
         WHEN 'MD5'
```

```
            THEN DBMS_CRYPTO.hash_md5
      END;
   l_in := utl_i18n.string_to_raw (p_in_val, 'AL32UTF8');
   l_hash_val := DBMS_CRYPTO.HASH (src => l_in, typ => l_hash_algo);
   l_ret := rawtohex(l_hash_val);
   RETURN l_ret;
END;
```

Рассмотрим пример ее использования:

```
SQL> SELECT get_hash_val ('Test')
  2  FROM DUAL
  3  /

GET_HASH_VAL('TEST')
----------------------------------------
640AB2BAE07BEDC4C163F679A746F7AB7FB5D1FA
```

Функция применяется для хеширования значения типа VARCHAR2 и возвращает значение типа VARCHAR2, которое можно сохранить и передать. По умолчанию функция использует алгоритм SHA-1, но может принимать и любой из остальных двух алгоритмов.

Другие области использования хеширования

Хеширование используется не только в криптографии, но и во многих других областях, например в веб-программировании и антивирусных средствах.

Веб-приложения не запоминают состояние: они не сохраняют соединение с сервером базы данных открытым на протяжении транзакции. Другими словами, для них не существует понятия «сеанс», и поэтому нет возможности блокирования данных в том виде, в каком она существует в традиционных приложениях. Так что простого способа определения того, что данные на веб-странице изменились, не существует. Но если сохранить вместе с данными хеш-значение, то потом можно вычислить его заново и сравнить с сохраненным значением. Если два значения не совпадут, это будет означать, что данные были изменены.

Хеширование также полезно при определении надежности данных. Представим себе вирус, который изменяет важные документы, хранящиеся в базе данных. Триггеру это не отловить. Однако если документ содержит хеш-значение, то, сравнив вычисленное значение с сохраненным, вы сможете определить, было ли искажено содержание документа и можно ли ему теперь доверять.

Код аутентификации сообщения (MAC) в Oracle 10g

Рассмотренный ранее метод хеширования весьма полезен, но имеет некоторые недостатки:

- Проверить подлинность переданных данных с помощью хеш-функции может кто угодно. Это может оказаться недопустимым для не-

которых систем повышенной безопасности, предполагающих, что аутентичность сообщения или данных проверяет только определенный получатель.

- Узнав алгоритм хеширования, злоумышленник может вычислить правильное хеш-значение и подменить им реальное, скрыв тем самым изменение данных.
- По причине, указанной в предыдущем пункте, хранение хеш-значения вместе с данными не представляется безопасным. Каждый, кто имеет привилегию на изменение таблицы, сможет изменить и хеш-значение. Аналогично некто может сгенерировать хеш-значение и изменить данные «в пути». Поэтому хеш-значение не должно сопровождать данные, а обязательно должно передаваться отдельно, что усложняет систему.

Справиться с этими недостатками помогает усовершенствованная реализация хеширования, в которой эксклюзивность механизма хеширования на стороне получателя удостоверяется паролем или ключом. Такое специальное хеш-значение называется *кодом аутентификации сообщения* (MAC – *Message Authentication Code*). Отправитель вычисляет MAC для данных, используя предопределенный ключ, который также известен получателю, но не отправляется вместе с данными. Отправитель отсылает получателю MAC вместе с данными, не разделяя их. После получения данных получатель также вычисляет значение MAC (используя тот же ключ) и сравнивает его со значением, пришедшим вместе с данными. Схематически данный механизм изображен на рис. 4.9.

Как и хеширование, MAC следует стандартным алгоритмам MD5 и SHA-1. При этом, как и в функции HASH, используемый алгоритм ука-

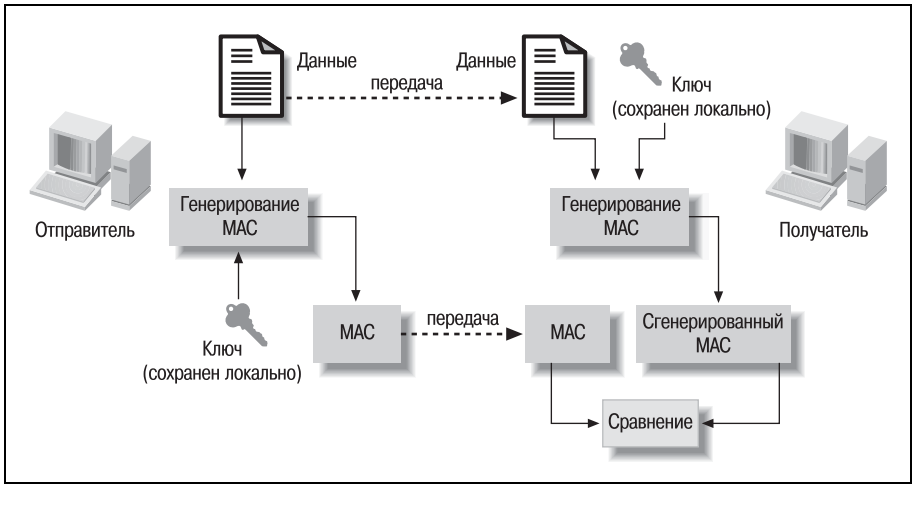


Рис. 4.9. *Использование кода аутентификации сообщения (Message Authentication Code)*

зывается в параметре typ. Требуется выбрать значение DBMS_CRYPTO.HMAC_MD5 или DBMS_CRYPTO.HMAC_SH1. В следующем примере значение MAC для входной строки вычисляется при помощи алгоритма SHA-1.

```
/* Файл на веб-сайте: get_sha1_mac_val.sql */
CREATE OR REPLACE FUNCTION get_sha1_mac_val (p_in RAW, p_key RAW)
   RETURN RAW
IS
   l_mac   RAW (4000);
BEGIN
   l_mac :=
      DBMS_CRYPTO.mac (src      => p_in,
                       typ      => DBMS_CRYPTO.hmac_sh1,
                       key      => p_key);
   RETURN l_mac;
END;
/
```

Можно также построить собственные вычисления MAC на основе нашей функции хеширования.

```
/* Файл на веб-сайте: get_mac_val.sql */
CREATE OR REPLACE FUNCTION get_mac_val (
   p_in_val      IN   VARCHAR2,
   p_key         IN   VARCHAR2,
   p_algorithm   IN   VARCHAR2 := 'SH1'
)
   RETURN VARCHAR2
IS
   l_mac_val    RAW (4000);
   l_key        RAW (4000);
   l_mac_algo   PLS_INTEGER;
   l_in         RAW (4000);
   l_ret        VARCHAR2 (4000);
BEGIN
   l_mac_algo :=
      CASE p_algorithm
         WHEN 'SH1'
            THEN DBMS_CRYPTO.hmac_sh1
         WHEN 'MD5'
            THEN DBMS_CRYPTO.hmac_md5
      END;
   l_in := utl_i18n.string_to_raw (p_in_val, 'AL32UTF8');
   l_key := utl_i18n.string_to_raw (p_key, 'AL32UTF8');
   l_mac_val := DBMS_CRYPTO.mac (src => l_in, typ => l_mac_algo, key=>l_key);
   l_ret := RAWTOHEX (l_mac_val);
   RETURN l_ret;
END;
```

Протестируем эту функцию, попробовав получить значение MAC для данных «Test Data» и ключа «Key».

```
SQL> SELECT get_mac_val ('Test Data','Key')
```

```
  2  FROM DUAL
  3  /

GET_MAC_VAL('TESTDATA','KEY')
----------------------------------------
8C36C24C767E305CD95415C852E9692F53927761
```

Для вычисления контрольной суммы требуется ключ, что делает данный метод более надежным, чем просто хеширование. Например, в банковских приложениях важна целостность таких данных, как номер социального страхования (Social Security number – SSN) клиента, владеющего счетом. Предположим, что таблица счетов ACCOUNTS выглядит следующим образом:

```
ACCOUNT_NO      NUMBER(10)
SSN             CHAR(9)
SSN_MAC         VARCHAR2(200)
```

При создании счета вычисляется значение MAC для поля SSN: используется предопределенный ключ, например «A Jolly Good Rancher». Столбец SSN_MAC обновляется следующим образом:

```
UPDATE accounts
   SET ssn_mac = get_mac_val (ssn, 'A Jolly Good Rancher')
 WHERE account_no = account_no;
```

Предположим теперь, что когда-нибудь впоследствии какой-то злоумышленник изменит поле SSN. Если бы поле SSN_MAC содержало хеш-значение столбца SSN, то злоумышленник мог бы самостоятельно вычислить хеш-значение и записать новое значение в этот столбец. Тогда при последующем вычислении хеш-значения для столбца SSN и его сравнении с сохраненным значением SSN_MAC значения бы совпали, и изменение данных выявить бы не удалось! Если столбец содержит не хеш-значение, а MAC-значение для столбца, то вычисление нового значения потребует знания ключа («A Jolly Good Rancher»). Не зная его, злоумышленник не сможет сгенерировать значение, равное MAC-значению, и изменение данных будет очевидным.

Создание реальной системы шифрования

Подведем итог. Применим полученные знания о шифровании и хешировании, построив реальную действующую систему шифрования.

В некоторых случаях требуется сравнение зашифрованных данных с входящими данными. Например, многие CRM-приложения (Customer Relationship Management – управление взаимоотношениями с клиентами) используют для уникальной идентификации клиентов такие атрибуты, как номера кредитных карт, номера паспортов и другие. Медицинским приложениям может потребоваться просмотр истории болезни для предложения плана лечения. Страховым компаниям может понадобиться просмотр диагнозов пациента для подтверждения

справедливости жалоб. Все эти данные хранятся в зашифрованном виде, поэтому простое сравнение входящих данных с сохраненными данными невозможно.

Существуют два способа обработки подобных ситуаций:

Зашифровать поступившие данные и сравнить их с сохраненными зашифрованными данными

Реализуется только в случае, если известен ключ шифрования. Если при шифровании был использован подход «один ключ для базы данных» (таблицы или схемы), то вы точно знаете, какой ключ следует применить для шифрования значений. Если был использован подход «новый ключ для каждой строки», то вам нужно знать, какой ключ следует применить для шифрования значения в каждой конкретной строке. Так что вы не сможете использовать данный способ.

Еще одним проблемным местом при использовании данного способа являются индексы. Если для данного зашифрованного столбца создан индекс, то этот индекс будет полезен при наличии предиката равенства (например, `ssn = encrypt ('123-45-6789')`). Запрос найдет в индексе зашифрованное значение строки «123-45-6789», а затем получит остальные значения данной строки таблицы. Если задано условие равенства, то в индексе производится поиск точного значения. Однако, если задать предикат подобия (например, `ssn like '123-%'`), индекс окажется бесполезным. Структура «b-tree» индекса устроена так, что рядом оказываются значения, у которых совпадают начальные символы. Индекс полезен для операции подобия, если значения заданы открытым текстом. В этом случае записи индекса для «123-45-6789» и «123-67-8945» оказались бы недалеко друг от друга. Но после шифрования такие значения могут превратиться в нечто подобное:

```
076A5703A745D03934B56F7500C1DCB4
178F45A983D5D03934B56F7500C1DCB4
```

Как видите, начальные символы зашифрованных значений сильно отличаются друг от друга, поэтому они попадут в разные части индекса. В результате поиск по индексу для определения местоположения в таблице окажется медленнее полного просмотра таблицы.

Расшифровать зашифрованные данные в каждой строке и сравнить их с соответствующими открытыми данными

Для тех, кто использует отдельный ключ для каждой строки, этот способ является единственно возможным. Но каждая операция дешифрования потребляет несколько драгоценных циклов ЦПУ и может повлиять на общую производительность базы данных.

Как же спроектировать систему так, чтобы сравнение с зашифрованными столбцами было наиболее эффективным? Фокус в том, чтобы выполнять сравнение не с зашифрованным значением, а с хеш-значе-

нием. Создание хеш-значения занимает значительно меньше времени, чем шифрование, и потребляет меньше циклов ЦПУ. Поскольку в результате хеширования некоторых данных всегда будет получено одно и то же хеш-значение, можно сохранить хеш-значение для конфиденциальных данных, создать хеш-значение для проверяемых на совпадение данных и сравнить его с сохраненным хеш-значением.

Предлагается следующая структура системы. Предположим, что у нас есть таблица CUSTOMERS, в которой хранятся номера кредитных карт, требующие шифрования. Вместо того чтобы сохранять в таблице CUSTOMERS номер кредитной карты, мы создадим две дополнительных таблицы (таблицы и связи между ними представлены на рис. 4.10).

Таблица CUSTOMERS

CUST_ID (первичный ключ)

CC (хеш-значение кредитной карты, но не сам реальный номер карты)

Таблица CC_MASTER

CC_HASH (первичный ключ)

ENC_CC# (зашифрованное значение номера кредитной карты)

Таблица CC_KEYS

CC_HASH (первичный ключ)

ENC_KEY (ключ шифрования, используемый для шифрования данного номера кредитной карты)

Незашифрованный номер кредитной карты нигде не хранится. Можно написать триггер, запускаемый перед вставкой (INSERT) или изменением (UPDATE) строки, который будет реализовывать такой псевдокод.

```
1  Вычислить хеш-значение
3  Искать это хеш-значение в таблице CC_MASTER
4  IF найдено THEN
5     Ничего не делать
6  ELSE
7     Сгенерировать ключ
8     Использовать этот ключ для получения зашифрованного значения
```

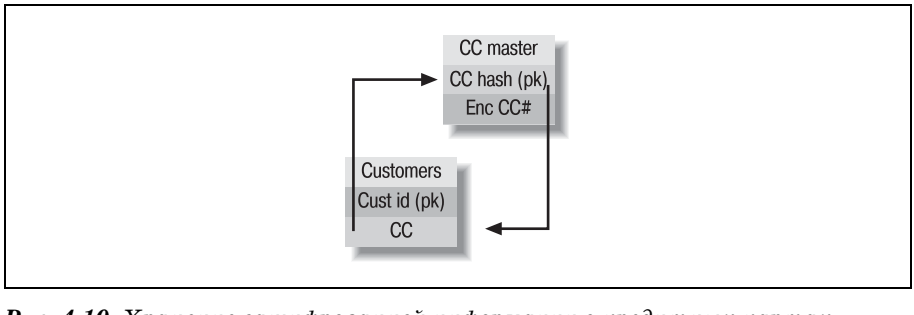


Рис. 4.10. *Хранение зашифрованной информации о кредитных картах*

```
      открытого номера кредитной карты
 9    Вставить в таблицу CC_KEYS запись с данным хеш-значением и ключом
10    Вставить в таблицу CC_MASTER запись с данным хеш-значением
      и зашифрованным значением.
11  END IF
```

Тем самым мы обеспечим отсутствие незашифрованного значения номера кредитной карты в базе данных. Приложения будут продолжать вставлять значения открытым текстом, но триггер будет заменять их хеш-значениями. Далее приводится возможный код такого триггера:

```
 1  CREATE OR REPLACE TRIGGER tr_aiu_customers
 2     BEFORE INSERT OR UPDATE
 3     ON customers
 4     FOR EACH ROW
 5  DECLARE
 6     l_hash   VARCHAR2 (64);
 7     l_enc    RAW (2000);
 8     l_key    RAW (2000);
 9  BEGIN
10     l_hash := get_hash_val (:NEW.cc);
11
12     BEGIN
13        SELECT cc_enc
14          INTO l_enc
15          FROM cc_master
16         WHERE cc_hash = l_hash;
17     EXCEPTION
18        WHEN NO_DATA_FOUND
19        THEN
20           BEGIN
21              l_key := get_key;
22              l_enc := get_enc_val (:NEW.cc, l_key);
23
24              INSERT INTO cc_master
25                          (cc_hash, cc_enc
26                          )
27                   VALUES (l_hash, l_enc
28                          );
29
30              INSERT INTO cc_keys
31                          (cc_hash, cc_key
32                          )
33                   VALUES (l_hash, l_key
34                          );
35           END;
36        WHEN OTHERS
37        THEN
38           RAISE;
39     END;
40
```

```
41     :NEW.cc := l_hash;
42 END;
```

Поясним ключевые моменты:

Строки	Описание
10	Сначала вычисляю хеш-значение для полученного открытым текстом от пользователя номера кредитной карты.
13–16	Проверяю, существует ли такое значение в таблице CC_MASTER.
21	Если совпадающее хеш-значение не найдено, это означает, что речь идет о новой кредитной карте. Для ее шифрования необходимо сначала сгенерировать ключ.
22	Использую этот ключ для шифрования открытого значения номера карты.
24–28	Вставляю зашифрованный номер кредитной карты в таблицу CC_MASTER.
30–34	Сохраняю ключ в таблице CC_KEYS.
41	Заменяю открытый номер кредитной карты соответствующим хеш-значением и сохраняю его.

Триггер заменяет открытое значение хеш-значением, поэтому изменение приложения не требуется. Программы, сравнивающие номера кредитных карт, будут искать совпадение с хеш-значением, а не с открытым или зашифрованным значениями. Используя такой триггер и представленные в главе функции, вы сможете создать собственную эффективную систему шифрования.

Заключение

В этой главе мы познакомились с шифрованием, управлением ключами, хешированием и некоторыми смежными вопросами. Давайте подведем некоторые итоги. Шифрование данных – это маскировка данных, выполняемая для того, чтобы скрыть их реальное значение. Для шифрования требуются входные данные, ключ шифрования и алгоритм шифрования. Существуют два базовых метода шифрования: асимметричное шифрование (шифрование с открытым ключом) – для шифрования и дешифрования используются разные ключи, и симметричное шифрование – для шифрования и дешифрования используется один и тот же ключ. Первый метод обычно используется при передаче данных и требует тщательной настройки, в то время как второй способ достаточно прост в применении.

Наиболее важным и сложным аспектом создания инфраструктуры шифрования является создание надежной и безопасной системы управления ключами, а не использование самих программных интерфейсов приложений. Для хранения ключей можно использовать раз-

нообразные возможности: можно хранить их в базе данных, в файловой системе или комбинировать эти типы хранилищ. Вы можете использовать один ключ для всей базы данных, отдельный ключ для каждой строки таблицы или какой-то промежуточный вариант. Можно использовать два разных ключа: обычный ключ, сохраненный в одном месте, и мастер-ключ, сохраненный в другом месте. Для шифрования данных будет использован не сохраненный ключ, а результат побитовой операции XOR для мастер-ключа и сохраненного ключа. Даже если один из этих ключей будет рассекречен, дешифровать данные удастся только после получения второго ключа.

В некоторых случаях нет необходимости в сокрытии данных, требуется лишь уверенность в том, что они не были изменены. Для этого существует криптографическое хеширование. Хеш-функция всегда возвращает одно и то же значение для заданного входного значения. То есть изменение вычисленного хеш-значения по отношению к исходному хеш-значению означает изменение исходных данных. Одна из разновидностей хеширования, MAC (код аутентификации сообщения), также использует ключ.

Oracle 10*g* Release 2 вводит новую возможность прозрачного шифрования – Transparent Database Encryption (TDE), обеспечивающую прозрачное шифрование и дешифрование данных перед их сохранением в полях данных. При использовании средств TDE конфиденциальные данные в файлах данных, архивных журнальных файлах и резервных копиях баз данных хранятся в зашифрованном виде, следовательно, кража таких файлов не приводит к рассекречиванию данных. Однако имейте в виду, что TDE не является полноценной системой шифрования, поскольку требует контроля над пользователями. Если вы хотите управлять доступом к дешифрованным значениям, то вам необходимо будет создать собственную инфраструктуру.

5

Контроль доступа на уровне строк

Технология RLS (row-level security, безопасность на уровне строк) позволяет задавать правила (политики) безопасности для таблиц базы данных (и отдельных типов операций над таблицами), ограничивающие для пользователя возможность чтения или изменения определенных строк в этих таблицах. Появившись в Oracle8*i*, эта технология стала очень полезным инструментом для администратора баз данных, поэтому в Oracle9*i* и Oracle 10*g* ее возможности были расширены. Функциональность RLS реализована в основном с помощью встроенного пакета `DBMS_RLS`.

В этой главе мы обсудим, как использовать пакет `DBMS_RLS` для создания и применения политик RLS в базе данных, и сравним возможности этой технологии в версиях Oracle9*i* и Oracle 10*g*. Рассмотрим также работу контекста приложения в связке с RLS и взаимодействие RLS с рядом других возможностей Oracle. Поскольку, вероятно, многие администраторы все еще используют Oracle9*i*, сначала рассмотрим средства RLS в этой версии, тем более что большая их часть перешла в Oracle 10*g*. Расширение возможностей RLS в Oracle 10*g* описано в разделе «RLS в Oracle 10*g*». Прежде чем углубляться в подробности работы RLS, мы рекомендуем вернуться на шаг назад и освежить свои представления о том, как осуществляются авторизация и доступ к базе данных.

Введение в RLS

Уже много лет Oracle обеспечивает безопасность на уровне таблиц и в некоторой степени на уровне столбцов. Пользователям могут быть выданы (или отозваны у них) привилегии на доступ к отдельным таблицам или столбцам. Определенным пользователям можно выдать права на вставку в один набор таблиц и на выборку данных из другого набора таблиц. Например, пользователь `John` может получить привилегию на операцию `SELECT` для таблицы `EMP`, принадлежащей пользова-

телю Scott, которая позволяет John'у получить любую строку этой таблицы, но не позволяет выполнить изменение, удаление или вставку. Привилегии на уровне объектов отвечают многим требованиям, но иногда они оказываются недостаточно детальными для выполнения разнообразных правил безопасности, которые часто налагаются на работу с корпоративными данными. Типичным примером являются демонстрационные таблицы Oracle, традиционно содержащие данные о сотрудниках. В таблице EMP хранятся данные обо всех сотрудниках компании, но руководителям отделов должна быть доступна информация только о работниках своего подразделения.

Ранее администраторы баз данных полагались на создаваемые поверх базовых таблиц представления, обеспечивающие безопасность на уровне строк. К сожалению, применение этого метода может привести к появлению огромного количества представлений, которые сложно оптимизировать и контролировать, особенно учитывая тот факт, что правила доступа к строкам могут со временем меняться.

Тут в дело вступает технология RLS. С ее помощью вы можете очень точно определить доступный пользователю набор строк таблицы, при этом контроль будет осуществляться PL/SQL-функциями, реализующими сложную логику правил. Управлять такими функциями гораздо проще, чем представлениями.

Технология RLS включает в себя три основных элемента.

Политика (policy)

Декларативная команда, которая определяет, как и когда следует применять ограничения пользовательского доступа для запросов, вставок, удалений, изменений или комбинаций перечисленных операций. Например, может потребоваться запретить для пользователя операции UPDATE, не ограничивая возможности выборки, или ограничить доступ к выбору данных из определенного столбца (например, сведения о зарплате, SALARY), не ограничивая выборку из остальных столбцов.

Функция политики безопасности (policy function)

Хранимая функция, которая вызывается в случае, когда выполняются условия, заданные в политике безопасности.

Предикат (predicate)

Строка, которая генерируется функцией политики безопасности, и которую Oracle прозрачно автоматически присоединяет в конец предложения WHERE выполняемых пользователем операторов SQL.

RLS автоматически применяет предикат к пользовательскому оператору SQL, вне зависимости от того, как этот оператор был выполнен. Предикат фильтрует строки на основании условия, определенного функцией политики безопасности. Если условие исключает все строки, которые не должны быть видны пользователю, то тем самым фактически обеспечивается безопасность на уровне строк. Ключевым моментом,

обеспечивающим высокую надежность и полноту технологии RLS, является то, что Oracle автоматически применяет предикат к пользовательскому SQL-оператору.

Зачем вам знать об RLS?

Исходя из сказанного, у вас могло сложиться впечатление, что RLS – это узкоспециализированная функция безопасности, которая вряд ли понадобится в каждодневной работе администратора базы данных. На самом деле, польза от применения RLS выходит за рамки обеспечения безопасности. Приведем краткий обзор причин, по которым администраторы баз данных находят применение RLS полезным (подробно эти причины будут рассмотрены далее в главе).

Повышение безопасности

Несомненно, основной целью RLS является повышение уровня безопасности внутри компании. Для многих компаний RLS обеспечивает соответствие новым правилам и инструкциям по обеспечению безопасности и конфиденциальности (например, Sarbanes-Oxley, HIPAA, Visa Cardholder Information Security Program), которые они обязаны выполнять. В наши дни безопасность – это не второстепенный вопрос, интересующий лишь затерянных где-то в глубине корпоративных джунглей аудиторов. Теперь это важная составляющая общего процесса проектирования и разработки системы. Сегодня каждый, начиная с самого младшего разработчика и заканчивая самым маститым администратором базы данных, должен быть хорошо знаком с инструментами и технологиями обеспечения безопасности. Oracle предоставляет множество дополнительных передовых возможностей и опций обеспечения безопасности, но технология RLS встроена в сервер базы данных Oracle и является первым средством, которое следует использовать для реализации политик безопасности. И новичок администрирования баз данных, и ветеран, за плечами которого годы разработки на PL/SQL, быстро поймут, что близкое знакомство с RLS поможет аккуратно интегрировать функции обеспечения безопасности в их базу данных.

Упрощение разработки и поддержки

RLS позволяет собрать всю логику политики безопасности в набор пакетов, содержащих хорошо структурированные функции PL/SQL. Даже если бы вы *смогли* реализовать имеющиеся требования безопасности на уровне строк при помощи представлений, *хотели* бы вы так поступить? Чтобы удовлетворить сложные бизнес-условия, требуются весьма витиеватые SQL-конструкции. По мере введения вашей компанией новых политик безопасности или усовершенствования старых, а также при вступлении в силу новых постановлений правительства вам нужно будет как-то переводить их на язык SQL для соответствующего изменения ваших представлений. Гораздо проще внести изменения в PL/SQL-функции, собранные

в небольшом количестве пакетов, и тем самым позволить Oracle автоматически применять ваши правила к определенным таблицам (вне зависимости от способа доступа).

Упрощение «коробочных» приложений

Простота разработки влечет за собой простоту адаптации «коробочных» приложений, разработанных третьими фирмами. Даже если бы задача изменения каждого запроса в приложении представлялась осуществимой, вам все равно не удалось бы проделать это для «коробочных» приложений в связи с отсутствием их исходных текстов. Потребовалась бы помощь поставщика программного обеспечения. Эта проблема особенно касается унаследованных систем: большинство компаний боится что-то менять в таких системах, даже если речь идет просто о добавлении дополнительного предиката. На помощь приходит технология RLS, не требующая внесения изменений в код. Вы можете проникнуть *под* код стороннего приложения, полностью минуя его логику, и добавить собственные политики для таблиц, с которыми работает этот код.

Управление доступом на запись

RLS обеспечивает гибкий быстрый и простой способ изменения уровня доступа к таблицам и представлениям с «только чтение» на «чтение и запись» на основании мандата пользователя. Встроенные команды администрирования Oracle позволяют определить табличные пространства в целом как доступные только для чтения или для чтения/записи. Технология RLS заполняет этот пробел, позволяя применить такие же правила к отдельным таблицам.

Простой пример

Давайте рассмотрим простой пример использования RLS. Будем работать с таблицей EMP в схеме HR, созданной при помощи сценария, поставляемого в составе программного обеспечения Oracle в файле *$ORACLE_HOME/sqlplus/demo/demobld.sql*.

```
SQL> DESC emp
 Name              Null?    Type
 ----------------- -------- ------------
 EMPNO             NOT NULL NUMBER(4)
 ENAME                      VARCHAR2(10)
 JOB                        VARCHAR2(9)
 MGR                        NUMBER(4)
 HIREDATE                   DATE
 SAL                        NUMBER(7,2)
 COMM                       NUMBER(7,2)
 DEPTNO                     NUMBER(2)
```

Таблица содержит 14 строк:

```
EMPNO ENAME      JOB          MGR HIREDATE     SAL   COMM  DEPTNO
----- ---------- ---------- ----- --------- ------ ------ -------
```

```
7369 SMITH      CLERK       7902 17-DEC-80    800              20
7499 ALLEN      SALESMAN    7698 20-FEB-81  1,600    300       30
7521 WARD       SALESMAN    7698 22-FEB-81  1,250    500       30
7566 JONES      MANAGER     7839 02-APR-81  2,975              20
7654 MARTIN     SALESMAN    7698 28-SEP-81  1,250  1,400       30
7698 BLAKE      MANAGER     7839 01-MAY-81  2,850              30
7782 CLARK      MANAGER     7839 09-JUN-81  2,450              10
7788 SCOTT      ANALYST     7566 09-DEC-82  3,000              20
7839 KING       PRESIDENT        17-NOV-81  5,000              10
7844 TURNER     SALESMAN    7698 08-SEP-81  1,500      0       30
7876 ADAMS      CLERK       7788 12-JAN-83  1,100              20
7900 JAMES      CLERK       7698 03-DEC-81    950              30
7902 FORD       ANALYST     7566 03-DEC-81  3,000              20
7934 MILLER     CLERK       7782 23-JAN-82  1,300              10
```

Зададим очень простое требование. Пусть необходимо, чтобы пользователи могли видеть данные только тех сотрудников, чья заработная плата не превышает 1500 долларов. Предположим, что пользователь вводит такой запрос:

```
SELECT * FROM emp;
```

Хотелось бы, чтобы средства RLS *прозрачно* преобразовывали этот запрос в такой:

```
SELECT * FROM emp WHERE sal <= 1500;
```

То есть при запросе пользователем данных из таблицы `EMP` Oracle (используя механизм RLS) будет автоматически применять необходимое ограничение. Чтобы все было именно так, надо сообщить Oracle об этих требованиях.

Сначала необходимо написать функцию, которая создает и возвращает такой предикат в виде строки. Простота данного требования позволяет использовать автономную функцию. В реальных приложениях вам придется определять функции предикатов и связанную с ними функциональность в пакетах. От имени пользователя `HR` создадим функцию `authorized_emps`.

```
CREATE OR REPLACE FUNCTION authorized_emps (
   p_schema_name   IN   VARCHAR2,
   p_object_name   IN   VARCHAR2
)
   RETURN VARCHAR2
IS
BEGIN
   RETURN 'SAL <= 1500';
END;
```

Обратите внимание, что оба аргумента (имя схемы и объекта) не используются внутри функции. Их наличие является требованием архитектуры RLS. Другими словами, каждая функция предиката должна получать эти два аргумента (ниже мы рассмотрим этот вопрос более подробно).

При выполнении функция возвращает наш предикат: SAL <= 1500. Проверим это, используя тестовый сценарий:

```
DECLARE
   l_return_string   VARCHAR2 (2000);
BEGIN
   l_return_string := authorized_emps ('X', 'X');
   DBMS_OUTPUT.put_line ('Return String = ' || l_return_string);
END;
```

Вывод будет таким:

```
Return String = SAL <= 1500
```

Имея функцию, возвращающую предикат, можно перейти к следующему шагу: созданию *политики безопасности*, также называемой *политикой RLS* или просто *политикой*. Эта политика определяет, когда и как предикат будет применяться к командам SQL. Для определения безопасности на уровне строк для таблицы EMP используем такой код:

```
1  BEGIN
2     DBMS_RLS.add_policy (object_schema  => 'HR',
3         object_name          => 'EMP',
4         policy_name          => 'EMP_POLICY',
5         function_schema      => 'HR',
6         policy_function      => 'AUTHORIZED_EMPS',
7         statement_types      => 'INSERT, UPDATE, DELETE, SELECT'
8         );
9  END;
```

Давайте внимательно посмотрим на то, что здесь происходит. Добавляется политика EMP_POLICY (строка 4) для таблицы EMP (строка 3), принадлежащей схеме HR (строка 2). Эта политика будет применять фильтр, задаваемый функцией AUTHORIZED_EMPS (строка 6), принадлежащей схеме HR (строка 5), при выполнении любым пользователем операции INSERT, UPDATE, DELETE или SELECT (строка 7).

Определив политику, мы можем сразу протестировать ее, выполнив запрос к таблице EMP:

```
SQL>SELECT * FROM hr.emp;

EMPNO ENAME      JOB         MGR HIREDATE     SAL       COMM DEPTNO
------ ---------- --------- ----- --------- ------ ---------- ------
  7369 SMITH      CLERK      7902 17-DEC-80    800                20
  7521 WARD       SALESMAN   7698 22-FEB-81  1,250        500     30
  7654 MARTIN     SALESMAN   7698 28-SEP-81  1,250       1400     30
  7844 TURNER     SALESMAN   7698 08-SEP-81  1,500          0     30
  7876 ADAMS      CLERK      7788 12-JAN-83  1,100                20
  7900 JAMES      CLERK      7698 03-DEC-81    950                30
  7934 MILLER     CLERK      7782 23-JAN-82  1,300                10
```

Как видите, выбрано только 7 *строк*, а не все 14. Присмотревшись, вы заметите, что во всех выбранных строках значение столбца SAL меньше или равно 1500, то есть соответствует функции предиката.

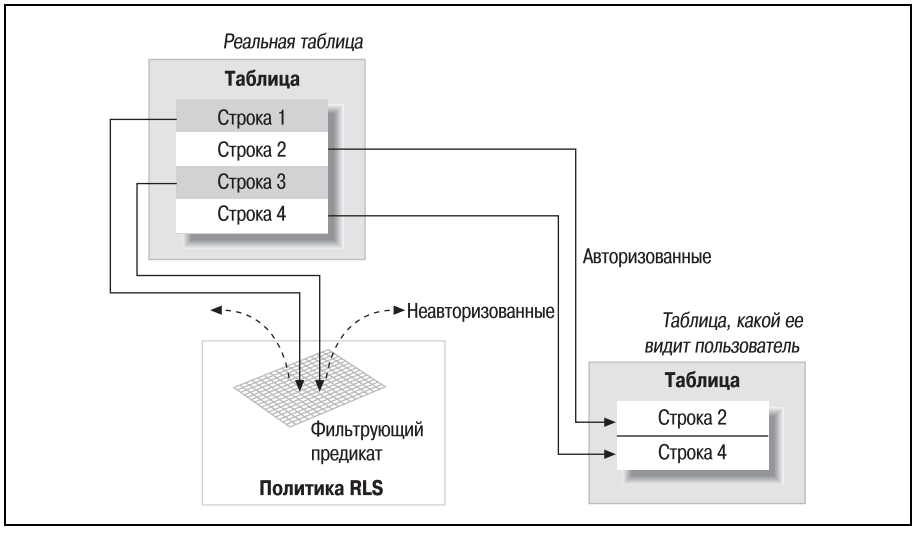


Рис. 5.1. *Работа фильтра, обеспечивающего безопасность на уровне строк*

Аналогично, если пользователи попытаются удалить или обновить все строки таблицы, им удастся выполнить операцию только для тех строк, видимость которых обеспечивает политика RLS:

```
SQL> DELETE hr.emp;
7 rows deleted.

SQL> UPDATE hr.emp SET comm = 100;
7 rows updated.
```

Oracle применяет такую фильтрацию на самом нижнем уровне, поэтому пользователи о ней даже не подозревают. Фактически они не знают о том, что им что-то недоступно, и это еще одно ценное свойство RLS с точки зрения безопасности.

Основной поток данных и реализуемая механизмом RLS фильтрация изображены на рис. 5.1.

Когда пользователь обращается к таблице, находящейся под контролем RLS, оператор SQL перехватывается и переписывается сервером базы данных с добавлением предиката, полученного от функции политики безопасности. Если функция политики безопасности возвращает корректное предложение-предикат, то он применяется к исходному оператору пользователя.

Политики не являются объектами схемы базы данных. Другими словами, они не принадлежат никакому пользователю. Любой пользователь, обладающий привилегией `EXECUTE` на пакет `DBMS_RLS`, может создать политику. Аналогично любой пользователь с привилегией `EXECUTE` может удалить любую политику. Поэтому необходимо очень внимательно подходить к выдаче прав

на работу с пакетом DBMS_RLS. Если кто-то выдаст привилегию EXECUTE на пакет для PUBLIC, ее надо немедленно отозвать.

Вы можете создавать функции политики безопасности любой сложности, описывающие практически любые требования к приложению. Однако все эти функции должны следовать нескольким правилам:

- Функция политики безопасности должна быть самостоятельной функцией в схеме или в составе пакета, но ни в коем случае не процедурой.
- Она должна возвращать значение типа VARCHAR2, которое будет использоваться как предикат.
- Функция должна иметь ровно два входных параметра, следующих в определенном порядке:
  a. имя схемы, которой принадлежит таблица, для которой определена политика;
  b. имя объекта (таблицы или представления), к которому применяется политика.

Для просмотра политик, определенных для таблицы, можно обратиться к представлению словаря данных DBA_POLICIES, которое отображает имя политики, имя объекта, для которого она определена (и его владельца), имя функции политики (и ее владельца) и многое другое. Полный перечень столбцов данного представления приведен в приложении А.

Если вы хотите удалить существующую политику RLS, то можете использовать программу DROP_POLICY из пакета DBMS_RLS. Примеры такого использования будут приведены в главе далее.

Кратко о политиках RLS

- Политика безопасности – это набор инструкций, которые применяются для обеспечения контроля над таблицей на уровне строк. Политика не является объектом схемы и не принадлежит ни одному из пользователей.
- Oracle использует политику для определения того, когда и как следует применять предикат ко всем операторам SQL, ссылающимся на таблицу.
- Предикат создается и возвращается функцией политики безопасности.

Использование RLS

После того как мы на примере познакомились с основами применения RLS, давайте перейдем к примерам, иллюстрирующим достоинства различных аспектов технологии RLS.

Проверка перед обновлением

Давайте немного изменим наш предыдущий пример. Вместо обновления столбца COMM пользователь теперь хочет изменить столбец SAL. SAL — это столбец, который используется в предикате, поэтому интересно будет посмотреть на результат.

```
SQL> UPDATE hr.emp SET sal = 1200;
7 rows updated.

SQL> UPDATE hr.emp SET sal = 1100;
7 rows updated.
```

Обновлены только семь строк, как и ожидалось. Теперь давайте изменим обновляемую сумму. В конце концов, все заслуживают повышения зарплаты.

```
SQL> UPDATE hr.emp SET sal = 1600;
7 rows updated.

SQL> UPDATE hr.emp SET sal = 1100;
0 rows updated.
```

Обратите внимание на последнюю операцию. Почему ни одна строка не была обновлена?

Все дело в первой операции обновления. Значения столбца SAL изменяются на 1600, и это новое значение не удовлетворяет условию предиката SAL <= 1500. То есть после первого обновления все строки становятся невидимыми для пользователя.

Так может возникнуть путаница: пользователь может выполнить для строк некоторый оператор SQL, в результате чего доступ к этим строкам будет изменен. При разработке приложения такая неустойчивость данных может вызвать ошибки или, по крайней мере, внести некий элемент непредсказуемости, усложняющий отладку. Чтобы решить проблему, используем еще один параметр процедуры ADD_POLICY, update_check. Давайте посмотрим, как установка этого параметра в значение TRUE повлияет на создание политики для таблицы.

```
BEGIN
   DBMS_RLS.add_policy (object_name  => 'EMP',
         policy_name          => 'EMP_POLICY',
         function_schema      => 'HR',
         policy_function      => 'AUTHORIZED_EMPS',
         statement_types      => 'INSERT, UPDATE, DELETE, SELECT',
         update_check         => TRUE
         );
END;
```

Если пользователь попытается выполнить то же самое обновление после того, как для таблицы создана эта новая политика, будет выдана ошибка:

```
SQL> UPDATE hr.emp SET sal = 1600;
update hr.emp set sal = 1600
          *
ERROR at line 1:
ORA-28115: policy with check option violation
```

Генерируется ошибка ORA-28115, потому что политика теперь предотвращает любые изменения значений тех столбцов, которые могли бы привести к изменению видимости строк для заданного предиката. Пользователи могут выполнять изменения значений других столбцов, так как они *не* влияют на видимость строк:

```
SQL> UPDATE hr.emp SET sal = 1200;
7 rows updated.
```

Я бы рекомендовал устанавливать параметр `update_check` в значение `TRUE` при каждом объявлении политики, с тем чтобы избежать непредсказуемого и, возможно, нежелательного поведения приложения в дальнейшем.

Статические политики RLS

Во всех рассмотренных ранее примерах использовалась *статическая* политика (значение, возвращаемое функцией предиката, не изменяется при изменении условий вызова функции).

Постоянство возвращаемого значения означает, что нет необходимости в выполнении функции для каждого запроса к таблице. Значение можно вычислить всего один раз, кэшировать его и затем использовать повторно столько раз, сколько потребуется. Чтобы воспользоваться этой возможностью, следует определить политику для RLS как *статическую*, передав значение `TRUE` аргументу `static_policy`:

```
 1  BEGIN
 2     DBMS_RLS.ADD_POLICY (
 3        object_name      => 'EMP',
 4        policy_name      => 'EMP_POLICY',
 5        function_schema  => 'HR',
 6        policy_function  => 'AUTHORIZED_EMPS',
 7        statement_types  => 'INSERT, UPDATE, DELETE, SELECT',
 8        update_check     => TRUE,
 9        static_policy    => TRUE
10     );
11  END;
```

По умолчанию параметр `static_policy` установлен в `FALSE`: в этом случае политика воспринимается как *динамическая* и вызывается для каждой операции над таблицей. Динамические политики будут описаны далее в разделе «Определение динамической политики». В Oracle 10*g* помимо статических и динамических поддерживаются и другие типы политик (подробнее поговорим об этом в разделе «Типы политик»).

Во многих ситуациях нужны именно статические политики. Возьмем, например, склад товаров, обслуживающий нескольких клиентов. В данном случае предикат может использоваться для предоставления доступа только к тем записям, которые относятся к данному клиенту. Например, в таблице BUILDINGS может быть столбец CUSTOMER_ID. К запросам будет присоединяться предикат CUSTOMER_ID = *customer_id*, где *customer_id* определяет пользователя, вошедшего в систему. При регистрации пользователя его клиентский идентификатор может быть извлечен специальным LOGON-триггером, а политика RLS может использовать этот идентификатор для определения того, какие строки следует показывать пользователю. Значение такого предиката не будет меняться на протяжении сеанса, поэтому имеет смысл установить параметр static_policy в значение TRUE.

Недостатки статических политик

Статические политики могут улучшить производительность, но могут и внести в приложение ошибки. Если предикат получается из или зависит от изменяющегося значения, такого как время, IP-адрес, идентификатор клиента или что-то еще, то разумнее задать динамическую политику. Покажем на примере, почему это именно так.

Давайте вернемся к нашей исходной функции политики, но предположим, что предикат зависит от изменяющегося значения, такого как секундная составляющая текущей временной метки (возможно, это не слишком реалистичный пример, но он удобен для пояснения).

```
SQL> CREATE TABLE trigger_fire
  2  (
  3     val NUMBER
  4  );

Table created.

SQL> INSERT INTO trigger_fire
  2  VALUES
  3  (1);

1 row created.

SQL> COMMIT;

Commit complete.

SQL> CREATE OR REPLACE FUNCTION authorized_emps (
  2     p_schema_name   IN   VARCHAR2,
  3     p_object_name   IN   VARCHAR2
  4  )
  5     RETURN VARCHAR2
  6  IS
  7     l_return_val   VARCHAR2 (2000);
  8     PRAGMA AUTONOMOUS_TRANSACTION;
  9  BEGIN
 10     l_return_val := 'SAL <= ' || TO_NUMBER (TO_CHAR (SYSDATE, 'ss')) * 100;
```

```
 11
 12     UPDATE trigger_fire
 13        SET val = val + 1;
 14
 15     COMMIT;
 16     RETURN l_return_val;
 17 END;
 18  /

Function created.
```

В этом примере функция берет секундную составляющую текущего времени (строка 10), умножает ее на 100 и возвращает предикат, который отображает значение столбца SAL, не превышающее полученное число. Секундная составляющая со временем изменяется, поэтому последующие исполнения данной функции приведут к получению различных результатов.

Определим для таблицы EMP политику, используя созданную функцию в качестве функции политики. Т. к. политика уже существует, то начинаем с ее удаления.

```
SQL> BEGIN
  2     DBMS_RLS.drop_policy (object_name => 'EMP', policy_name =>
                                                        'EMP_POLICY');
  3  END;
  4  /

PL/SQL procedure successfully completed.

SQL> BEGIN
  2     DBMS_RLS.add_policy (object_name         => 'EMP',
  3                          policy_name         => 'EMP_POLICY',
  4                          function_schema     => 'HR',
  5                          policy_function     => 'AUTHORIZED_EMPS',
  6                          statement_types     => 'INSERT, UPDATE,
                                                     DELETE, SELECT',
  7                          update_check        => TRUE,
  8                          static_policy       => FALSE
  9                         );
 10  END;
 11  /

PL/SQL procedure successfully completed.
```

Теперь проверим политику. Пусть пользователь Ленни попытается определить количество служащих в таблице.

```
SQL> SELECT COUNT(*) FROM hr.emp;

  COUNT(*)
----------
         0
```

Таблица находится под контролем RLS, поэтому вызывается функция политики, возвращающая предикат, который применяется к запросу. Предикат зависит от секундной составляющей текущей временной метки, так что его значение лежит в диапазоне от 0 до 60. В данном конкретном случае значение оказалось таким, что ни одна из строк не соответствовала условию предиката.

Функция политики обновляет столбец VAL таблицы TRIGGER_FIRE, поэтому я могу определить, сколько раз функция была вызвана. От имени пользователя HR выбираем значение VAL из таблицы TRIGGER_FIRE.

```
SQL> SELECT * FROM trigger_fire;

       VAL
----------
         3
```

Функция политики была вызвана дважды: один раз на этапе синтаксического разбора, а второй – на этапе выполнения, поэтому значение увеличилось на 2 единицы (с начального значения 1). Ленни еще раз выполняет запрос, желая узнать количество служащих.

```
SQL> SELECT COUNT(*) FROM hr.emp

  COUNT(*)
----------
        10
```

На этот раз функция политики возвращает предикат, которому удовлетворяют 10 записей таблицы. Снова проверяем значение VAL в таблице TRIGGER_FIRE.

```
SQL> SELECT * FROM trigger_fire;

       VAL
----------
         5
```

Значение увеличилось на 2 (с 3), это означает, что функция политики выполнялась несколько раз. Вы можете повторить проверку сколько угодно раз, чтобы убедиться в том, что функция политики выполняется каждый раз при выполнении операции над таблицей.

Теперь объявим политику как статическую и повторим тест. Операции *изменения* политики ни в RLS, ни в API не существует, поэтому удалим и пересоздадим ее.

```
SQL> BEGIN
  2     DBMS_RLS.drop_policy (object_name => 'EMP', policy_name =>
                                                    'EMP_POLICY');
  3  END;
  4  /

PL/SQL procedure successfully completed.
```

Посмотрим, что изменится.

```
 1  BEGIN
 2     DBMS_RLS.add_policy (object_name          => 'EMP',
 3                          policy_name          => 'EMP_POLICY',
 4                          function_schema      => 'HR',
 5                          policy_function      => 'AUTHORIZED_EMPS',
 6                          statement_types      => 'INSERT, UPDATE,
                                                     DELETE, SELECT',
 7                          update_check         => TRUE,
 8                          static_policy        => TRUE
 9                         );
10  END;
11  /

PL/SQL procedure successfully completed.

SQL> -- Сбросим значение в таблице TRIGGER_FIRE
SQL> UPDATE trigger_fire SET val = 1;

1 row updated.

SQL> COMMIT;

Commit complete.
```

От имени пользователя Ленни выберем количество строк таблицы:

```
SQL> SELECT COUNT(*) FROM hr.emp;

  COUNT(*)
----------
         8
```

Теперь проверим значение столбца VAL таблицы TRIGGER_FIRE от имени HR:

```
SQL> SELECT * FROM trigger_fire;

       VAL
----------
         2
```

Значение увеличилось на единицу, потому что функция политики была выполнена единожды, а не дважды, как в прошлый раз. Пусть пользователь Ленни еще несколько раз повторит выборку из таблицы EMP.

```
SQL> SELECT COUNT(*) FROM hr.emp;

  COUNT(*)
----------
         8

SQL> SELECT COUNT(*) FROM hr.emp;

  COUNT(*)
----------
         8

SQL> SELECT COUNT(*) FROM hr.emp;

  COUNT(*)
----------
         8
```

Все время возвращается *одно и то же* число. Почему? Потому что функция политики была выполнена только один раз, затем предикат был кэширован. Больше функция политики ни разу не исполнялась, и предикат не менялся. Подтвердим наше предположение проверкой значения VAL в таблице TRIGGER_FIRE от имени пользователя HR.

```
SQL> SELECT * FROM trigger_fire;

       VAL
----------
         2
```

Имеем все то же значение 2; никаких изменений с момента первого вызова функции. Полученные выходные данные подтверждают, что функция политики не вызывалась при последующих выборках из таблицы EMP.

Объявив политику как *статическую*, мы фактически указали, что функция политики должна быть выполнена только один раз, затем политика должна повторно использовать изначально созданный предикат, даже если его значение могло измениться с течением времени. Такое поведение может привести к неожиданным последствиям для вашего приложения, поэтому следует использовать статические политики с большой осторожностью. Единственный случай, в котором вы, вероятно, захотите использовать именно статические политики, – когда функция возвращает определенный предикат, не зависящий от каких бы то ни было переменных, за исключением тех, которые были установлены с началом сеанса и больше не изменялись (например, имена пользователей).

Использование прагмы

Еще один способ обеспечить исполнение необходимой нам логики – использовать пакетную функцию с объявлением *прагмы* для подавления некоторого набора операций над базой данных. Рассмотрим спецификацию пакета.

```
CREATE OR REPLACE PACKAGE rls_pkg
AS
   FUNCTION authorized_emps (
      p_schema_name   IN   VARCHAR2,
      p_object_name   IN   VARCHAR2
   )
      RETURN VARCHAR2;

   PRAGMA RESTRICT_REFERENCES (authorized_emps, WNDS, RNDS, WNPS, RNPS);
END;
/
```

И тело пакета.

```
CREATE OR REPLACE PACKAGE BODY rls_pkg
AS
   FUNCTION authorized_emps (
```

```
        p_schema_name   IN   VARCHAR2,
        p_object_name   IN   VARCHAR2
    )
        RETURN VARCHAR2
    IS
        l_return_val   VARCHAR2 (2000);
    BEGIN
        l_return_val :=
            'SAL <= ' || TO_NUMBER (TO_CHAR (SYSDATE, 'ss')) * 100;
        RETURN l_return_val;
    END;
END;
/
```

В этой спецификации пакета определена *прагма*, вводящая следующие уровни строгости для данной функции:

WNDS

Write No Database State – не записывать состояние базы данных.

RNDS

Read No Database State – не читать состояние базы данных.

WNPS

Write No Package State – не записывать состояние пакета.

RNPS

Read No Package State – не читать состояние пакета.

При компиляции тела данного пакета прагма будет нарушена, и компиляция будет прервана с выдачей сообщения об ошибке.

```
Warning: Package Body created with compilation errors.

Errors for PACKAGE BODY RLS_PKG:

LINE/COL ERROR
-------- -----------------------------------------------------------------
2/4      PLS-00452: Subprogram 'AUTHORIZED_EMPS' violates its associated
         pragma
```

Объявление прагмы защищает вас от возникновения потенциально ошибочных ситуаций. Однако в любом случае разумно использовать статические политики в обусловленных ситуациях, как в рассмотренном ранее примере со складом.

При создании статической политики убедитесь в том, что возвращаемый функцией политики предикат не изменяет значение в течение сеанса.

Определение динамической политики

В предыдущем разделе мы говорили о политике, которая возвращает строку предиката, являющуюся константой, например `SAL <= 1500`.

В реальной жизни такие сценарии используются не слишком часто, за исключением некоторых специализированных приложений, таких как склады. В большинстве случаев необходима фильтрация пользователей, выдающих запросы. Например, в приложении управления персоналом может потребоваться, чтобы пользователь видел только свои собственные записи, а не все записи таблицы. Это *динамическое* требование, так как оно должно проверяться для каждого пользователя, входящего в систему. Функцию политики можно переписать следующим образом:

```
 1 CREATE OR REPLACE FUNCTION authorized_emps (
 2    p_schema_name   IN   VARCHAR2,
 3    p_object_name   IN   VARCHAR2
 4 )
 5    RETURN VARCHAR2
 6 IS
 7    l_return_val   VARCHAR2 (2000);
 8 BEGIN
 9    l_return_val := 'ENAME = USER';
10    RETURN l_return_val;
11 END;
12 /
```

В строке 9 предикат сравнивает значение столбца ENAME со значением USER, то есть с именем текущего зарегистрированного пользователя. Если пользователь Martin (если помните, Martin – это имя одного из служащих в таблице EMP) входит в систему и выполняет выборку из таблицы, он видит всего одну строку – свою собственную.

```
SQL> CONN martin/martin
Connected.

SQL> SELECT * FROM hr.emp;

EMPNO ENAME      JOB          MGR HIREDATE     SAL   COMM DEPTNO
------ ---------- --------- ------ --------- ------ ------ ------
  7654 MARTIN     SALESMAN    7698 28-SEP-81  1,250  1,400     30
```

Давайте теперь несколько расширим задачу: пусть Martin видит не только собственную запись, но и записи для всего своего отдела. В этом случае функция политики будет такой:

```
 1  CREATE OR REPLACE FUNCTION authorized_emps (
 2    p_schema_name   IN   VARCHAR2,
 3    p_object_name   IN   VARCHAR2
 4 )
 5    RETURN VARCHAR2
 6 IS
 7    l_deptno       NUMBER;
 8    l_return_val   VARCHAR2 (2000);
 9 BEGIN
10    SELECT deptno
```

```
11       INTO l_deptno
12       FROM emp
13      WHERE ename = USER;
14
15     l_return_val := 'DEPTNO = ' || l_deptno;
16     RETURN l_return_val;
17  END;
18  /
```

Необходимо сделать еще один шаг. В предыдущем фрагменте кода функция осуществляла выборку из таблицы EMP (строки 10–13). Однако таблица защищена политикой RLS, функция которой принадлежит пользователю HR. Будучи выполненной с привилегиями пользователя HR, она не найдет ни одной строки, так как сотрудника с именем HR не существует, что делает предикат некорректным. Существуют два способа решения этой проблемы:

- Выдать пользователю HR специальную привилегию, с тем чтобы к нему не применялись политики RLS, или
- Внутри функции политики указать, что если вызывающий ее пользователь является владельцем схемы, то для него проверку проводить не следует.

При использовании первого подхода нет необходимости в изменении функции политики. От имени администратора базы данных выдаем специальную привилегию пользователю HR:

```
GRANT EXEMPT ACCESS POLICY TO hr;
```

Тем самым для пользователя HR из приложения будут удалены все политики. Этот подход следует использовать с большой осторожностью, так как будут отменены все политики – для всех таблиц. Осознавая, какую брешь это создает в модели безопасности, мы бы порекомендовали не применять данный подход для обычных владельцев схем.

Второй подход заключается в создании специальной схемы, например RLSOWNER, в которой будут созданы все политики RLS, и которой будут принадлежать все функции политики безопасности. Только этот пользователь, и никакие другие, получает системную привилегию EXEMPT ACCESS POLICY. Функция политики принадлежит RLSOWNER, поэтому PL/SQL-блок создания политики будет выглядеть следующим образом (его сможет запускать любой пользователь, имеющий привилегии EXECUTE на пакет DBMS_RLS):

```
1  BEGIN
2     DBMS_RLS.add_policy (object_name => 'EMP',
3            policy_name          => 'EMP_POLICY',
4            function_schema      => 'RLSOWNER',
5            policy_function      => 'AUTHORIZED_EMPS',
6            statement_types      => 'INSERT, UPDATE, DELETE, SELECT',
7            update_check         => TRUE
8            );
```

```
 9  END;
10  /
```

Если используется второй подход, то логика обхода фильтра для владельца схемы должна быть реализована в функции политики. Этот блок кода также может быть запущен любым пользователем, имеющим привилегии EXECUTE на пакет DBMS_RLS.

```
 1  CREATE OR REPLACE FUNCTION authorized_emps (
 2     p_schema_name   IN   VARCHAR2,
 3     p_object_name   IN   VARCHAR2
 4  )
 5     RETURN VARCHAR2
 6  IS
 7     l_deptno       NUMBER;
 8     l_return_val   VARCHAR2 (2000);
 9  BEGIN
10     IF (p_schema_name = USER)
11     THEN
12        l_return_val := NULL;
13     ELSE
14        SELECT deptno
15          INTO l_deptno
16          FROM emp
17         WHERE ename = USER;
18
19        l_return_val := 'DEPTNO = ' || l_deptno;
20     END IF;
21
22     RETURN l_return_val;
23  END;
24  /
```

Эта версия функции очень похожа на предыдущие. Новые строки выделены жирным шрифтом (строка 10). Здесь проверяется, является ли вызывающий пользователь владельцем таблицы. Если это так, возвращается NULL (строка 12). Значение NULL в предикате, возвращенном функцией, эквивалентно полному отсутствию политики, то есть строки не фильтруются.

Пользователь Martin выполняет тот же запрос, что и раньше:

```
SQL> SELECT * FROM hr.emp;

 EMPNO ENAME      JOB          MGR HIREDATE     SAL   COMM DEPTNO
------ ---------- --------- ------ --------- ------ ------ ------
  7499 ALLEN      SALESMAN    7698 20-FEB-81  1,600    300     30
  7521 WARD       SALESMAN    7698 22-FEB-81  1,250    500     30
  7654 MARTIN     SALESMAN    7698 28-SEP-81  1,250  1,400     30
  7698 BLAKE      MANAGER     7839 01-MAY-81  2,850            30
  7844 TURNER     SALESMAN    7698 08-SEP-81  1,500      0     30
  7900 JAMES      CLERK       7698 03-DEC-81    950            30

6 rows selected.
```

Обратите внимание, что все возвращенные строки относятся к отделу пользователя Martin – 30.

Как видите, функция политики является важнейшим элементом создания политики RLS. Политика будет применять к строкам фильтры на основании значения любого предиката, возвращенного функцией, необходима лишь синтаксическая корректность. При помощи функций политики безопасности вы можете создавать весьма замысловатые и сложные предикаты.

Точно так же можно применять фильтры RLS для любой таблицы базы данных. Например, можно создать политику для таблицы DEPT:

```
 1  BEGIN
 2     DBMS_RLS.add_policy (object_schema   => 'HR',
 3             object_name          => 'DEPT',
 4             policy_name          => 'DEPT_POLICY',
 5             function_schema      => 'RLSOWNER',
 6             policy_function      => 'AUTHORIZED_EMPS',
 7             statement_types      => 'SELECT, INSERT, UPDATE, DELETE',
 8             update_check         => TRUE
 9           );
10  END;
11  /
```

Та же самая функция, AUTHORIZED_EMPS, используется в качестве функции политики. Функция возвращает предикат DEPTNO = *deptno*, поэтому она может использоваться в таблице DEPT, как и в любой другой таблице, содержащей столбец DEPTNO.

Таблица, в которой нет столбца DEPTNO, может содержать другой столбец, являющийся внешним ключом для таблицы EMP. Например, в таблице BONUS есть столбец ENAME, который связан с таблицей EMP. Перепишем нашу функцию политики следующим образом:

```
CREATE OR REPLACE FUNCTION allowed_enames (
   p_schema_name   IN   VARCHAR2,
   p_object_name   IN   VARCHAR2
)
   RETURN VARCHAR2
IS
   l_deptno       NUMBER;
   l_return_val   VARCHAR2 (2000);
   l_str          VARCHAR2 (2000);
BEGIN
   IF (p_schema_name = USER)
   THEN
      l_return_val := NULL;
   ELSE
      SELECT deptno
        INTO l_deptno
        FROM hr.emp
```

```
        WHERE ename = USER;

      l_str := '(';

      FOR emprec IN (SELECT ename
                       FROM hr.emp
                      WHERE deptno = l_deptno)
      LOOP
         l_str := '''' || emprec.ename || ''',';[1]
      END LOOP;

      l_str := RTRIM (l_str, ',');
      l_str := ')';
      l_return_val := 'ENAME IN ' || l_str;
   END IF;

   RETURN l_return_val;
END;
/
```

Определяем политику для таблицы BONUS посредством следующей функции политики; тем самым политика RLS будет введена в действие и для нее:

```
BEGIN
   DBMS_RLS.add_policy (object_schema  => 'HR',
          object_name          => 'BONUS',
          policy_name          => 'BONUS_POLICY',
          function_schema      => 'RLSOWNER',
          policy_function      => 'ALLOWED_ENAMES',
          statement_types      => 'SELECT, INSERT, UPDATE, DELETE',
           update_check         => TRUE
          );
END;
/
```

Так можно определить политики RLS для всех связанных таблиц базы данных, находящихся в зависимости от данной. В связи с тем, что описанная функциональность обеспечивает персональное представление таблиц базы данных на основании характеристик пользователя или каких-то других параметров (например, времени дня или IP-адреса), ее называют *виртуальной частной базой данных* (*Virtual Private Database* (VPD)).

Повышение производительности

Предположим, что наши требования опять изменились (что совсем неудивительно, не правда ли?). Теперь нужно создать политику так, что-

[1] Здесь значение переменной l_str будет перезаписываться при каждой итерации цикла вместо конкатенации. Необходимо писать l_str := l_str || '''' || emprec.ename ||''','; . – *Примеч. науч. ред.*

бы все пользователи и подразделения были доступны для пользователя, являющегося руководителем. Для других пользователей должны быть видны только сотрудники их подразделения. Для соответствия таким требованиям функция политики должна выглядеть следующим образом:

```
CREATE OR REPLACE FUNCTION authorized_emps (
   p_schema_name   IN   VARCHAR2,
   p_object_name   IN   VARCHAR2
)
   RETURN VARCHAR2
IS
   l_deptno       NUMBER;
   l_return_val   VARCHAR2 (2000);
   l_mgr          BOOLEAN;
   l_empno        NUMBER;
   l_dummy        CHAR (1);
BEGIN
   IF (p_schema_name = USER)
   THEN
      l_return_val := NULL;
   ELSE
      SELECT DISTINCT deptno, empno
                 INTO l_deptno, l_empno
                 FROM hr.emp
                WHERE ename = USER;

      BEGIN
         SELECT '1'
           INTO l_dummy
           FROM hr.emp
          WHERE mgr = l_empno AND ROWNUM < 2;

         l_mgr := TRUE;
      EXCEPTION
         WHEN NO_DATA_FOUND
         THEN
            l_mgr := FALSE;
         WHEN OTHERS
         THEN
            RAISE;
      END;

      IF (l_mgr)
      THEN
         l_return_val := NULL;
      ELSE
         l_return_val := 'DEPTNO = ' || l_deptno;
      END IF;
   END IF;

   RETURN l_return_val;
END;
```

Посмотрите, какой сложной стала выборка данных. Эта сложность, несомненно, увеличит время отклика (а в реальных приложениях логика, конечно, будет значительно сложнее). Можно ли упростить код и повысить производительность?

Конечно. Обратимся к первому требованию: проверке того, является ли служащий руководителем. В приведенном выше коде такая проверка проводилась по таблице EMP, но ведь смена должности с неруководящей на руководящую происходит не очень часто. Кроме того, возможен карьерный рост руководителя, но статус его как руководителя при этом не меняется. В реальности звание руководителя похоже на атрибут, с которым пользователь входит в систему и который не изменяется в течение сеанса. Поэтому, если при входе в систему передать базе данных сведения о том, что пользователь является руководителем, то функции политики уже не придется проводить такую проверку.

Как передать подобное значение? Можно использовать глобальные переменные. Можно обозначить статус руководителя значением Y или N и создать пакет для хранения переменной.

```
CREATE OR REPLACE PACKAGE mgr_check
IS
   is_mgr CHAR (1);
END;
```

Функция политики будет такой:

```
CREATE OR REPLACE FUNCTION authorized_emps (
   p_schema_name   IN   VARCHAR2,
   p_object_name   IN   VARCHAR2
)
   RETURN VARCHAR2
IS
   l_deptno       NUMBER;
   l_return_val   VARCHAR2 (2000);
BEGIN
   IF (p_schema_name = USER)
   THEN
      l_return_val := NULL;
   ELSE
      SELECT DISTINCT deptno
                 INTO l_deptno
                 FROM hr.emp
                WHERE ename = USER;

      IF (mgr_check.is_mgr = 'Y')
      THEN
         l_return_val := NULL;
      ELSE
         l_return_val := 'DEPTNO = ' || l_deptno;
      END IF;
   END IF;
```

```
   RETURN l_return_val;
END;
```

Обратите внимание на то, насколько меньший объем кода требуется для проверки статуса руководителя. Статус проверяется по глобальной переменной пакета. Эта переменная должна быть задана на этапе входа пользователя в систему, и эту задачу отлично может выполнить триггер базы данных AFTER LOGON.

```
CREATE OR REPLACE TRIGGER tr_set_mgr
   AFTER LOGON ON DATABASE
DECLARE
   l_empno   NUMBER;
   l_dummy   CHAR (1);
BEGIN
   SELECT DISTINCT empno
              INTO l_empno
              FROM hr.emp
             WHERE ename = USER;

   SELECT '1'
     INTO l_dummy
     FROM hr.emp
    WHERE mgr = l_empno AND ROWNUM < 2;

   rlsowner.mgr_check.is_mgr := 'Y';
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      rlsowner.mgr_check.is_mgr := 'N';
   WHEN OTHERS
   THEN
      RAISE;
END;
/
```

Триггер устанавливает значение переменной пакета для обозначения руководящего статуса служащего, которое затем попадает в функцию политики. Давайте проведем быстрый тест. Подключившись от имени пользователя King (который является руководителем) и пользователя Martin (который таковым не является), посмотрим, как это работает.

```
SQL> CONN martin/martin
Connected.

SQL> SELECT COUNT(*) FROM hr.emp;

  COUNT(*)
----------
    262145

SQL> CONN king/king
Connected.
```

```
SQL> SELECT COUNT(*) FROM hr.emp;

  COUNT(*)
----------
    589825
```

Запрос для Martin извлек не всех пользователей (как и предполагалось), в то время как запрос для King извлек все строки.

Использование переменных пакета часто может приводить к значительному повышению производительности. В первом примере, когда проверка статуса руководителя проводилась внутри функции политики, запрос выполнялся 1,99 секунды. Использование глобальных переменных приводит к сокращению времени выполнения запроса до 1,32 секунды, что значительно лучше.

Контроль доступа к таблице

Области применения RLS не ограничиваются обеспечением безопасности и упрощением процесса разработки приложений. Технология RLS также чрезвычайно полезна в случаях, когда необходимо в зависимости от ряда условий менять состояние таблицы с «доступна только для чтения» на «доступна для чтение и записи». Без применения RLS администратор базы данных может изменить разрешение на доступ для целого табличного пространства, но не для его отдельных таблиц. При этом табличное пространство невозможно сделать доступным только для чтения при наличии хотя бы одной активной транзакции. Поскольку, возможно, не найдется такого периода времени, в течение которого в базе данных не выполняется ни одной транзакции, то перевод табличного пространства в состояние «только для чтения» окажется невозможным. В таких ситуациях единственно возможным решением будет использование технологии RLS.

Если быть до конца честным, то следует сказать, что RLS не выполняет реального перевода таблицы в состояние «только для чтения», а позволяет нам эмулировать такой перевод, запретив любые попытки изменения содержимого таблицы. Проще всего это сделать, применив к любому оператору `UPDATE`, `DELETE` и `INSERT` заведомо ложный предикат (всегда вычисляемый как `FALSE`), например `1=2`.

Приведем пример обеспечения доступа только для чтения к таблице `EMP` при помощи использования этой наипростейшей предикатной функции:

```
CREATE OR REPLACE FUNCTION make_read_only (
   p_schema_name   IN   VARCHAR2,
   p_object_name   IN   VARCHAR2
)
   RETURN VARCHAR2
IS
BEGIN
```

```
   -- Только владелец предикатной функции политики может изменять
      данные в таблице.
   IF (p_schema_name = USER)
   THEN
      RETURN NULL;
   ELSE
      RETURN '1=2';
   END IF;
END;
```

На основе этой функции можно создать политику RLS для таблицы EMP в отношении изменяющих данные операторов DML: INSERT, UPDATE и DELETE.

```
BEGIN
   DBMS_RLS.add_policy (object_name          => 'EMP',
                        policy_name          => 'EMP_READONLY_POLICY',
                        function_schema      => 'HR',
                        policy_function      => 'MAKE_READ_ONLY',
                        statement_types      => 'INSERT, UPDATE, DELETE',
                        update_check         => TRUE
                       );
END;
```

Обратите внимание, что параметр statement_types не включает в себя оператор SELECT, так как использование этого оператора не ограничивается.

Теперь текущий пользователь, не являющийся владельцем функции политики, сможет только выбирать данные из таблицы EMP:

```
SQL> SHOW user
USER is "MARTIN"

SQL> DELETE hr.emp;
0 rows deleted.

SQL> SELECT COUNT(*) FROM hr.emp;

  COUNT(*)
----------
        14
```

Когда приходит время снова разрешить запись в таблицу, мы просто отменяем политику.

```
BEGIN
   DBMS_RLS.enable_policy (object_name      => 'EMP',
                           policy_name      => 'EMP_READONLY_POLICY',
                           ENABLE           => FALSE
                          );
END;
/
```

Теперь и не-владельцы функции политики могут успешно выполнять DML-операции над таблицей.

Фактически таблица никогда не переходит в состояние доступа только для чтения. Политика гарантирует, что при выполнении пользователем DML-оператора для таблицы строки изменены не будут. Т. к. никакого сообщения об ошибке не выдается, а политика игнорирует DML-операторы, то следует внимательно анализировать код приложений, использующих такую функциональность. Программисты могут неумышленно и ошибочно интерпретировать отсутствие ошибки как успешное выполнение операции DML.

Технология RLS позволяет не только переводить таблицы в состояние доступа только для чтения или для чтения и записи по требованию, но и делать это динамически в зависимости от любых пользовательских условий. Например, вы можете написать функцию политики, которая делает таблицы доступными только для чтения с 5 часов вечера и до 9 часов утра для всех пользователей за исключением владельца пакетного задания (BATCHUSER), а для BATCHUSER таблица будет доступна только для чтения с 9 часов утра до 5 часов вечера. Тело такой функции могло бы выглядеть следующим образом:

```
BEGIN
 IF (p_schema_name = USER)
 THEN
    l_return_val := NULL;
 ELSE
    l_hr := TO_NUMBER (TO_CHAR (SYSDATE, 'HH24'));

    IF (USER = 'BATCHUSER')
    -- можно перечислить всех пользователей, которые в дневное время
       будут иметь доступ только для чтения.
    THEN
       IF (l_hr BETWEEN 9 AND 17)
       THEN
          -- перевести таблицу в состояние доступа только для чтения
          l_return_val := '1=2';
       ELSE
          l_return_val := NULL;
       END IF;
    ELSE
       -- можно перечислить всех пользователей, которые в ночное время
          будут иметь доступ только для чтения
       IF (l_hr >= 17 AND l_hr <= 9)
       THEN
          -- перевести таблицу в состояние доступа только для чтения
          l_return_val := '1=2';
       ELSE
          l_return_val := NULL;
       END IF;
    END IF;
 END IF;

 RETURN l_return_val;
END;
```

Используя временные метки, вы можете обеспечить детальный доступ к таблице. Можно использовать анализ и других различных атрибутов (например, IP-адрес, тип аутентификации, клиентские данные, терминал, пользователь операционной системы и многое другое). Нужно лишь получить соответствующую переменную из системного контекста (SYS_CONTEXT; мы поговорим об этой функции далее в этой главе) сеанса и проверить ее значение. Предположим, например, что пользователю King (который является президентом компании) разрешено видеть все записи при выполнении двух таких условий:

- Подключение осуществляется с компьютера KINGLAP с фиксированным IP-адресом (192.168.1.1) и из домена Windows NT с именем ACMEBANK.
- В Windows регистрируется пользователь King.

Теперь функция политики будет выглядеть так:

```
CREATE OR REPLACE FUNCTION emp_policy (
   p_schema_name   IN   VARCHAR2,
   p_object_name   IN   VARCHAR2
)
   RETURN VARCHAR2
IS
   l_deptno       NUMBER;
   l_return_val   VARCHAR2 (2000);
BEGIN
   IF (p_schema_name = USER)
   THEN
      l_return_val := NULL;
   ELSIF (USER = 'KING')
   THEN
      IF (
             -- проверить имя машины клиента
             SYS_CONTEXT ('USERENV', 'HOST') = 'ACMEBANK\KINGLAP'
          OR
             -- проверить имя пользователя ОС
             SYS_CONTEXT ('USERENV', 'OS_USER') = 'king'
          OR
             -- проверить IP-адрес
             SYS_CONTEXT ('USERENV', 'IP_ADDRESS') = '192.168.1.1'
         )
      THEN
         -- KING прошел все проверки; разрешить неограниченный доступ.
         l_return_val := NULL;
      ELSE
         -- вернуть обычный предикат
         l_return_val := 'SAL <= 1500';
      END IF;
   ELSE  -- все остальные пользователи
      l_return_val := 'SAL <= 1500';
   END IF;
```

```
    RETURN l_return_val;
END;
```

Здесь использована встроенная функция SYS_CONTEXT для получения атрибутов *контекста*. Использованию системных контекстов будет посвящен раздел «Контексты приложения». Пока вам нужно знать лишь то, что вызов функции возвращает имя клиентского терминала, с которого осуществлен вход в систему. Другие строки с вызовом функции также возвращают соответствующие значения.

Функцию SYS_CONTEXT можно использовать для получения разнообразной информации о пользовательском подключении. На основе такой информации вы можете настроить функцию политики таким образом, чтобы фильтр отвечал вашим специфическим требованиям. Полный перечень атрибутов, которые можно получить вызовом функции SYS_CONTEXT, можно найти в справочном руководстве по Oracle SQL.

RLS в Oracle 10g

В этом разделе описаны новые возможности RLS, появившиеся в версии Oracle 10*g*.

Применение RLS к отдельным столбцам

Давайте вернемся к примеру с приложением HR, использованному в предыдущих разделах. Была создана политика, реализующая следующие требования: просмотр всех записей разрешен только пользователю King; все остальные пользователи могут видеть только информацию о сотрудниках своего отдела. В некоторых ситуациях такая политика может оказаться слишком строгой.

Предположим, что мы хотим сделать так, чтобы пользователи не могли «пронюхать», у кого из сотрудников какая зарплата. Рассмотрим два запроса:

```
SELECT empno, sal FROM emp;

SELECT empno FROM emp;
```

Первый запрос выводит данные о зарплате – те самые данные, которые мы хотим защитить. В этом случае следует отображать сведения только о служащих, входящих в отдел пользователя. Второй запрос выводит только номера служащих. Следует ли выполнять фильтрацию, отображая номера только для служащих того отдела, к которому относится пользователь?

Ответ зависит от того, какая политика безопасности принята в каждой конкретной организации. Вполне возможно, было бы удобнее, если бы второй запрос выводил всех сотрудников, вне зависимости от отдела, в котором они работают.

В Oracle9*i* невозможно настроить RLS так, чтобы выполнить наше новое требование. В версии Oracle 10*g* у процедуры ADD_POLICY появляется

необходимый для этого новый параметр sec_relevant_cols. Я хочу изменить предыдущий сценарий так, чтобы фильтр применялся только в том случае, когда выбираются данные из столбцов SAL и COMM. Новая политика будет такой (новый параметр выделен жирным шрифтом):

```
BEGIN
   DBMS_RLS.drop_policy (object_schema      => 'HR',
                         object_name        => 'EMP',
                         policy_name        => 'EMP_POLICY'
                        );
   --
   DBMS_RLS.add_policy (object_schema          => 'HR',
            object_name            => 'EMP',
            policy_name            => 'EMP_POLICY',
            function_schema        => 'RLSOWNER',
            policy_function        => 'AUTHORIZED_EMPS',
            statement_types        => 'INSERT, UPDATE, DELETE, SELECT',
            update_check           => TRUE,
            sec_relevant_cols      => 'SAL, COMM'
            );
END;
```

После ввода в действие новой политики запросы пользователя Martin будут выполнены по-другому.

```
SQL> -- "безопасный" запрос, выбирается только EMPNO
SQL> SELECT empno FROM hr.emp;

... data displayed ...

14 rows selected.

SQL> -- запрос, содержащий SAL
SQL> SELECT sal FROM hr.emp;

... data displayed ...

6 rows selected.
```

При попытке выборки данных из столбца SAL политика RLS предотвращает отображение всех строк, отфильтровывая строки, в которых значение DEPTNO отлично от 30 (от номера отдела пользователя (Martin), выполняющего запрос).

Политика будет применяться для выбранных столбцов не только при их появлении в списке SELECT, но и при любой (явной или неявной) ссылке на такие столбцы. Рассмотрим, например, запрос:

```
SQL> SELECT   deptno, COUNT (*)
  2      FROM hr.emp
  3     WHERE sal > 0
  4  GROUP BY deptno;

    DEPTNO   COUNT(*)
---------- ----------
        30          6
```

Столбец SAL упоминается в предложении WHERE. В дело вступает политика RLS, что приводит к тому, что отображаются только записи для отдела 30. Давайте рассмотрим еще один пример, в котором я попытаюсь *вывести* значение SAL.

```
SQL> SELECT *
  2    FROM hr.emp
  3   WHERE deptno = 10;

no rows selected
```

Явной ссылки на столбец SAL нет, но *неявно* на него ссылается предложение SELECT *, поэтому политика RLS отфильтровывает все строки, не относящиеся к отделу 30. Запрос был вызван для отдела 10, поэтому ни одной строки не возвращено.

Теперь давайте несколько изменим условия. В прошлый раз мы добивались того, чтобы не отображались значения столбца SAL для тех строк, видеть которые пользователь не авторизован. Однако получилось так, что мы подавили вывод всей строки, а не только значения отдельного столбца. Сформулируем новое требование: следует маскировать только столбец, а не всю строку (то есть все остальные столбцы должны отображаться). Можно ли этого добиться?

Эту задачу легко решить при помощи еще одного параметра процедуры ADD_POLICY, sec_relevant_cols_opt. Единственное, что нужно сделать, — это пересоздать политику, установив этот параметр в константу DBMS_RLS.ALL_ROWS.

```
BEGIN
   DBMS_RLS.drop_policy (object_schema      => 'HR',
                         object_name        => 'EMP',
                         policy_name        => 'EMP_POLICY'
                        );
   DBMS_RLS.add_policy (object_schema              => 'HR',
                        object_name                => 'EMP',
                        policy_name                => 'EMP_POLICY',
                        function_schema            => 'RLSOWNER',
                        policy_function            => 'AUTHORIZED_EMPS',
                        statement_types            => 'SELECT',
                        update_check               => TRUE,
                        sec_relevant_cols          => 'SAL, COMM',
                        sec_relevant_cols_opt      => DBMS_RLS.all_rows
                       );
END;
```

Если Martin теперь выполнит тот же запрос, результат будет другим (в последующем выводе вместо значений NULL отображается «?»):

```
SQL> SET NULL ?
SQL> SELECT * FROM hr.emp ORDER by deptno;

EMPNO ENAME      JOB          MGR HIREDATE     SAL   COMM DEPTNO
------ ---------- --------- ------ --------- ------ ------ ------
```

```
7782 CLARK      MANAGER     7839 09-JUN-81 ?      ?          10
7839 KING       PRESIDENT ?      17-NOV-81 ?      ?          10
7934 MILLER     CLERK       7782 23-JAN-82 ?      ?          10
7369 SMITH      CLERK       7902 17-DEC-80 ?      ?          20
7876 ADAMS      CLERK       7788 12-JAN-83 ?      ?          20
7902 FORD       ANALYST     7566 03-DEC-81 ?      ?          20
7788 SCOTT      ANALYST     7566 09-DEC-82 ?      ?          20
7566 JONES      MANAGER     7839 02-APR-81 ?      ?          20
7499 ALLEN      SALESMAN    7698 20-FEB-81  1,600     300     30
7698 BLAKE      MANAGER     7839 01-MAY-81  2,850 ?           30
7654 MARTIN     SALESMAN    7698 28-SEP-81  1,250  1,400     30
7900 JAMES      CLERK       7698 03-DEC-81    950 ?           30
7844 TURNER     SALESMAN    7698 08-SEP-81  1,500       0     30
7521 WARD       SALESMAN    7698 22-FEB-81  1,250     500     30

14 rows selected.
```

Как видите, выведены *все* 14 строк, причем со значениями всех столбцов, только значения SAL и COMM заменены на NULL в тех строках, которые пользователь не должен видеть (то есть относящиеся не к отделу 30).

Новые параметры процедуры ADD_POLICY позволяют создать такую политику RLS, чтобы выводить все строки, скрывая лишь секретные значения. До выхода версии Oracle 10*g* для решения этой задачи пришлось бы использовать представления, и все было бы значительно сложнее.

В версии Oracle 10*g* Release 2 можно применять средства RLS даже к оператору CREATE INDEX. Для этого задайте INDEX в качестве значения параметра statement_types в процедуре ADD_POLICY.

Последнюю возможность следует применять с особой осторожностью, так как в некоторых случаях возможны неожиданные результаты. Предположим, например, что Martin выдает следующий запрос:

```
SQL> SELECT COUNT(1), AVG(sal) FROM hr.emp;
COUNT(SAL)   AVG(SAL)
---------- ----------
        14 1566.66667
```

Получены следующие данные: служащих – 14, средняя зарплата равна 1566. Но на самом деле средняя зарплата вычислена только для тех 6 служащих, данные о которых доступны пользователю Martin, а не для всех 14! Возможна путаница: какие значения следует считать корректными? Ведь если тот же запрос выдаст владелец схемы HR, то результат будет другим.

```
SQL> CONN hr/hr
Connected.

SQL> SELECT COUNT(1), AVG(sal) FROM hr.emp;
COUNT(SAL)   AVG(SAL)
---------- ----------
        14 2073.21429
```

При анализе результатов необходимо помнить о том, что они зависят от пользователя, выполнившего запрос. В противном случае в приложении могут возникнуть ошибки, которые сложно отследить.

Типы политик

Вероятно, наиболее важным усовершенствованием технологии RLS в Oracle 10*g* (в дополнение к имеющейся динамической политике) является поддержка новых типов политик, обеспечивающих повышение производительности.

Для начала давайте вспомним, чем статические политики отличаются от динамических. Если политика относится к динамическому типу, то функция политики выполняется для создания строки предиката каждый раз, когда политика предусматривает ограничение доступа к таблице. Использование динамической политики гарантирует «свежесть» предиката, но повторное выполнение функции политики приводит к дополнительным накладным расходам, которые могут быть весьма значительными. В большинстве случаев в повторном выполнении функции политики нет необходимости, так как предикат не изменяется в рамках сеанса (мы говорили об этом при обсуждении статических политик).

С точки зрения производительности лучше всего было бы создать функцию политики так, чтобы она выполнялась повторно при изменении какого-то определенного значения. Oracle 10*g* обеспечивает такую возможность: при изменении контекста приложения, от которого зависит программа, политика инициирует повторное выполнение функции; в противном случае функция повторно не вызывается. В последующих разделах мы поговорим о том, как это работает.

Как и в Oracle9*i*, в Oracle 10*g* вы можете установить параметр `static_policy` процедуры `ADD_POLICY` в значение `TRUE` (для выбора статической политики) или `FALSE` (для выбора динамической политики). Если данный параметр установлен в `TRUE`, то значение нового параметра Oracle 10*g*, `policy_type`, устанавливается в `DBMS_RLS.STATIC`. Если `static_policy` равен `FALSE`, то `policy_type` устанавливается в `DBMS_RLS.DYNAMIC`. По умолчанию `static_policy` принимает значение `TRUE`.

Выбор статической или динамической политики осуществляется так же, как и в Oracle9*i*, но Oracle 10*g* поддерживает несколько дополнительных типов политик. Вы можете выбрать их, указав соответствующее значение для параметра `policy_type` процедуры `ADD_POLICY`. Приведем перечень новых значений параметра.

Контекстно-зависимая политика (context-sensitive)

```
DBMS_RLS.CONTEXT_SENSITIVE
```

Разделяемая контекстно-зависимая политика (shared context sensitive)

```
DBMS_RLS.SHARED_CONTEXT_SENSITIVE
```

Разделяемая статическая политика (shared static)

```
DBMS_RLS.SHARED_STATIC
```

Новые типы политики значительно улучшают производительность, но вносят некоторые побочные эффекты, присущие статическим политикам (о которых мы говорили выше).

Разделяемые статические политики

Разделяемые статические политики очень похожи на статические политики. Разница лишь в том, что одна функция политики используется в политиках для нескольких объектов. В предыдущем примере вы видели, как функция `authorized_emps` использовалась в качестве функции политики для таблицы `DEPT` и таблицы `EMP`. Аналогично можно определить для обеих таблиц не только общую функцию, но и общую политику. Такая политика будет называться *разделяемой*. Если при этом она будет статической, то называться такая политика будет разделяемой статической политикой, а задаваться будет установкой параметра `policy_type` в константу `DBMS_RLS.SHARED_STATIC`. Используя данный тип политики, создадим общую политику для двух наших таблиц.

```
BEGIN
   DBMS_RLS.drop_policy (object_schema      => 'HR',
                         object_name        => 'DEPT',
                         policy_name        => 'EMP_DEPT_POLICY'
                        );
   DBMS_RLS.add_policy (object_schema  => 'HR',
          object_name          => 'DEPT',
          policy_name          => 'EMP_DEPT_POLICY',
          function_schema      => 'RLSOWNER',
          policy_function      => 'AUTHORIZED_EMPS',
          statement_types      => 'SELECT, INSERT, UPDATE, DELETE',
          update_check         => TRUE,
          policy_type          => DBMS_RLS.shared_static
          );
   DBMS_RLS.add_policy (object_schema => 'HR',
          object_name          => 'EMP',
          policy_name          => 'EMP_DEPT_POLICY',
          function_schema      => 'RLSOWNER',
          policy_function      => 'AUTHORIZED_EMPS',
          statement_types      => 'SELECT, INSERT, UPDATE, DELETE',
          update_check         => TRUE,
          policy_type          => DBMS_RLS.shared_static
          );
END;
```

Объявляя единую политику для двух таблиц, мы в действительности сообщаем базе данных о том, что результат функции политики следует кэшировать, а затем повторно использовать кэшированное значение.

Контекстно-зависимые политики

Мы уже говорили, что статические политики, несмотря на их эффективность, могут представлять определенную опасность, т. к. отсутствие повторного выполнения функции политики может приводить к неожиданным и нежелательным последствиям. Поэтому Oracle предлагает еще один тип политики: контекстно-зависимые политики, которые повторно выполняют функцию политики только при изменении контекста приложения в рамках сеанса (см. раздел «Контексты приложения» далее в главе). Рассмотрим блок кода, определяющий такую политику:

```
BEGIN
   DBMS_RLS.drop_policy (object_schema      => 'HR',
                         object_name        => 'EMP',
                         policy_name        => 'EMP_DEPT_POLICY'
                        );
   DBMS_RLS.add_policy (object_schema => 'HR',
          object_name          => 'EMP',
          policy_name          => 'EMP_DEPT_POLICY',
          function_schema      => 'RLSOWNER',
          policy_function      => 'AUTHORIZED_EMPS',
          statement_types      => 'SELECT, INSERT, UPDATE, DELETE',
          update_check         => TRUE,
          policy_type          => DBMS_RLS.context_sensitive
        );
END;
```

Использование контекстно-зависимой политики (`DBMS_RLS.CONTEXT_SENSITIVE`) может значительно увеличить производительность. В следующем фрагменте кода встроенная функция `DBMS_UTILITY.GET_TIME` вычисляет затраченное время с точностью до сотых долей секунды.

```
DECLARE
   l_start   PLS_INTEGER;
   l_count   PLS_INTEGER;
BEGIN
   l_start := DBMS_UTILITY.get_time;

   SELECT COUNT (*)
     INTO l_count
     FROM hr.emp;

   DBMS_OUTPUT.put_line (
      'Elapsed time = '
      || TO_CHAR (DBMS_UTILITY.get_time - l_start)
      );
END;
```

Выполним этот код, применяя все перечисленные в таблице типы политик. Как видите, чисто статическая политика оказывается самой быстрой (требуется всего одно выполнение функции политики). Контекстно-зависимая политика также значительно быстрее, чем полностью динамическая версия.

Тип политики	Время отклика (в сотых долях секунды)
Динамическая	133
Контекстно-зависимая	84
Статическая	37

Разделяемые контекстно-зависимые политики

Разделяемые контекстно-зависимые политики похожи на контекстно-зависимые. Отличие в том, что одна и та же политика используется для нескольких объектов, как и в случае разделяемых статических политик.

Отладка RLS

RLS – это сложная технология, взаимодействующая с разнообразными элементами архитектуры Oracle. Некорректное проектирование или неправильное применение пользователями может привести к возникновению ошибок. К счастью, для большей части ошибок RLS формирует подробный файл трассировки (в каталоге, определенном параметром инициализации базы данных `USER_DUMP_DEST`). В этом разделе мы поговорим о том, как отслеживать операции RLS и разрешать ошибочные ситуации.

Интерпретация ошибок

Чаще других вам будет встречаться ошибка ORA-28110: «Policy function or package has error» (ошибка функции политики или пакета), с которой легко справиться. Проблема в том, что при компиляции функции политики возникла одна или несколько ошибок. Исправив ошибки компиляции и заново скомпилировав функцию (или пакет, содержащий данную функцию), вы решите проблему.

Переход к типам политик, доступным в Oracle 10g

При переходе с Oracle9*i* на Oracle 10*g* я рекомендую действовать следующим образом:

1. Сначала использовать тип по умолчанию (динамический).
2. По завершении обновления попытаться пересоздать политику как контекстно-зависимую и тщательно проверить результаты для всех возможных сценариев, чтобы избежать возможных проблем, которые может повлечь кэширование.
3. Наконец, преобразовать в статические те политики, для которых это возможно, и так же тщательно проверить результаты.

Могут также возникнуть ошибки во время выполнения, такие как необработанное исключение, несоответствие типов данных или ситуации, когда объем выбранных данных значительно превышает размер переменной, в которую они выбираются. В этих случаях Oracle порождает ошибку ORA-28112: невозможно выполнить функцию политики, и генерирует файл трассировки. Определить причину ошибки можно, проанализировав файл трассировки, хранящийся в каталоге, заданном в параметре инициализации USER_DUMP_DEST. Рассмотрим фрагмент файла трассировки:

```
Policy function execution error:
Logon user     : MARTIN
Table/View     : HR.EMP
Policy name    : EMP_DEPT_POLICY
Policy function: RLSOWNER.AUTHORIZED_EMPS
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at "RLSOWNER.AUTHORIZED_EMPS", line 14
ORA-06512: at line 1
```

Видно, что ошибка возникла, когда пользователь Martin выполнял запрос. Функция политики выбрала более одной строки. Исследовав функцию политики, обнаруживаем такой фрагмент:

```
SELECT deptno
INTO l_deptno
FROM hr.emp
WHERE ename = USER;
```

Похоже, что существует несколько сотрудников с именем Martin, поэтому и было извлечено несколько строк, что привело к возникновению проблемы. Решение состоит в обработке ошибки через исключение или использовании другого предиката для получения номера отдела.

Еще одна ошибка, ORA-28113: ошибка в предикате политики, возникает, когда функция политики некорректно строит предложение предиката. Как и в предыдущем случае, формируется файл трассировки. Рассмотрим такой фрагмент:

```
Error information for ORA-28113:
Logon user     : MARTIN
Table/View     : HR.EMP
Policy name    : EMP_DEPT_POLICY
Policy function: RLSOWNER.AUTHORIZED_EMPS
RLS predicate  :
DEPTNO = 10,
ORA-00907: missing right parenthesis
```

Мы видим, что функция политики возвращает следующий предикат:

```
DEPTNO = 10,
```

SQL-запрос получается синтаксически некорректным, поэтому политика не работает и запрос пользователя Martin не выполняется. Следу-

ет исправить логику функции политики так, чтобы возвращаемый предикат был корректной строкой.

Операции в прямом режиме

Если вы используете операции в прямом режиме (*direct-path operations*), например прямую загрузку в SQL*Loader, прямую вставку (Direct Path Insert) с использованием подсказки APPEND (INSERT /*+ APPEND */ INTO …) или прямой экспорт, то при использовании средств RLS могут возникнуть проблемы. Эти операции минуют уровень SQL, поэтому политика RLS для таких таблиц не вызывается, и требования безопасности не проверяются. Как решить эту проблему?

С экспортом все просто. Вот что произойдет при прямом экспорте таблицы EMP (DIRECT=Y), которая защищена одной или несколькими политиками RLS.

```
About to export specified tables via Direct Path ...
EXP-00080: Data in table "EMP" is protected. Using conventional mode.
EXP-00079: Data in table "EMP" is protected. Conventional path may only
           be exporting partial table.
```

Экспорт проведен успешно, но как видите, использован не прямой путь, как нам хотелось, а *обычный* (*conventional path*). При выполнении операции экспорта политики RLS применяются к таблице: пользователь может экспортировать не все строки, а только те, доступ к которым ему разрешен.

Успешно выполненный экспорт для таблицы, защищенной политикой RLS, может вызвать ложное впечатление о произведенном экспорте всех строк. Не забывайте о том, что экспортируются только те строки, выборка которых разрешена пользователю.

Попытавшись выполнить прямую загрузку в таблицу, защищенную политикой RLS (используя SQL*Loader или Direct Path Insert), получим ошибку.

```
SQL> INSERT /*+ APPEND */
  2  INTO hr.EMP
  3  SELECT *
  4  FROM hr.emp
  5  WHERE rownum < 2;
FROM hr.emp
       *
ERROR at line 4:
ORA-28113: policy predicate has error
```

Сообщение об ошибке говорит само за себя: ошибка в предикате. Решить проблему можно, временно отменив политику для таблицы EMP или осуществив экспорт от имени пользователя, обладающего системной привилегией EXEMPT ACCESS POLICY.

Проверка перезаписи запроса

При отладке может возникнуть необходимость проверить модифицированный политикой RLS оператор SQL. Мы ведь не хотим действовать наугад или полагаться на удачу. Увидеть измененный оператор можно при помощи представления словаря данных или задав событие.

Представление словаря данных

Можно использовать представление словаря данных V$VPD_POLICY. «VPD» в названии представления означает Virtual Private Database – виртуальная частная база данных (название, которое иногда используют для технологии безопасности на уровне строк). Это представление отображает все изменения запроса, выполненные политикой RLS.

```
SQL> SELECT sql_text, predicate, policy, object_name
  2  FROM v$sqlarea , v$vpd_policy
  3  WHERE hash_value = sql_hash
  4  /

SQL_TEXT                    PREDICATE
--------------------------- -------------------------------
POLICY                       OBJECT_NAME
--------------------------- -----------------------------
select count(*) from hr.emp    DEPTNO = 10
EMP_DEPT_POLICY                EMP
```

Столбец SQL_TEXT содержит точный текст SQL-оператора, выданного пользователем, а столбец PREDICATE – предикат, сформированный функцией политики и примененный к запросу. Используя данное представление, вы можете увидеть пользовательские операторы и примененные к ним предикаты.

Трассировка событий

Можно также задать событие внутри сеанса и проанализировать файл трассировки. Прежде чем выполнить запрос, пользователь Martin выполняет дополнительную команду создания события.

```
SQL> ALTER SESSION SET EVENTS
    '10730 trace name context forever, level 12';
Session altered.

SQL> SELECT COUNT(*) FROM hr.emp;
```

После завершения запроса файл трассировки появится в каталоге, указанном параметром инициализации базы данных USER_DUMP_DEST. Он будет содержать следующие данные:

```
Logon user     : MARTIN
Table/View     : HR.EMP
Policy name    : EMP_DEPT_POLICY
Policy function: RLSOWNER.AUTHORIZED_EMPS
```

```
RLS view :
SELECT  "EMPNO","ENAME","JOB","MGR","HIREDATE","SAL","COMM","DEPTNO" FROM
"HR"."EMP" "EMP" WHERE (DEPTNO = 10)
```

Используя любой из предложенных способов, вы сможете увидеть, каким образом были перезаписаны запросы пользователя.

Взаимодействие RLS с другими функциями Oracle

Как и любая другая мощная технология, RLS обладает своим набором возможных проблем и сложностей. В этом разделе будет описано взаимодействие между RLS и несколькими другими функциями Oracle.

Ссылочное ограничение целостности

Если для таблицы, на которую наложена политика RLS, действует ссылочное ограничение целостности, указывающее на родительскую таблицу, на которую также наложена политика RLS, то обработка ошибок Oracle может создавать проблемы для безопасности. Предположим, что для таблицы DEPT определена политика RLS, разрешающая пользователю видеть данные только о своем отделе. В этом случае запрос «всех строк» таблицы DEPT выведет всего одну строку:

```
SQL> CONN martin/martin
Connected.

SQL> SELECT * FROM hr.dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
```

Однако для таблицы EMP политика RLS не определена, поэтому пользователь может свободно выбирать из нее данные. Так что он вполне может получить информацию о том, что отделов существует несколько.

```
SQL> SELECT DISTINCT deptno FROM hr.emp;

    DEPTNO
----------
        10
        20
        30
```

Таблица EMP имеет ссылочное ограничение целостности для столбца DEPTNO, которое ссылается на столбец DEPTNO таблицы DEPT.

Пользователь может видеть подробные данные только для отдела 10, к которому он принадлежит, при этом он знает о существовании других отделов. Предположим теперь, что он попытается изменить таблицу EMP, установив номер отдела в 50.

```
SQL> UPDATE hr.emp
  2  SET deptno = 50
```

```
  3  WHERE empno = 7369;
update hr.emp
*
ERROR at line 1:
ORA-02291: integrity constraint (HR.FK_EMP_DEPT) violated -
           parent key not found
```

Ошибка указывает на нарушение ограничения целостности. И оно действительно имеет место, так как в таблице DEPT нет строки со значением DEPTNO, равным 50. Средства Oracle выполнили свою работу, но в результате пользователь получил больше знаний о таблице DEPT, чем это подразумевалось политикой безопасности.

При некоторых обстоятельствах выявление *отсутствия* данных может быть столь же серьезным нарушением безопасности, как и отображение данных, *присутствующих* в таблице.

Тиражирование

При симметричном тиражировании (*multi-master replication*) схемам получателя и распространителя разрешено выбирать данные из таблиц без ограничений. Следовательно, вам придется или изменить функцию политики так, чтобы возвращать для таких пользователей предикат NULL, или предоставить им системную привилегию EXEMPT ACCESS POLICY.

Материализованные представления

При определении материализованных представлений необходимо убедиться в том, что владелец схемы материализованных представлений имеет неограниченный доступ к базовым таблицам. В противном случае запрос, определяющий материализованное представление, вернет только строки, удовлетворяющие предикату, что неверно. Как и в случае с тиражированием, следует изменить функцию политики так, чтобы возвращать для такого пользователя предикат NULL или предоставить ему системную привилегию EXEMPT ACCESS POLICY.

Контексты приложения

До этого момента все разговоры о безопасности на уровне строк велись в предположении, что предикат (то есть условие, ограничивающее доступ к строкам таблицы) не изменяется с момента входа пользователя в систему. Введем новое требование: пользователи должны видеть записи о сотрудниках в зависимости не от фиксированных номеров отделов, а от списка привилегий, специально поддерживаемых для этих целей. Таблица EMP_ACCESS хранит сведения о том, какому пользователю какая информация о сотрудниках доступна.

```
SQL> DESC emp_access
 Name              Null?    Type
 ----------------- -------- ------------
```

```
USERNAME                       VARCHAR2(30)
DEPTNO                         NUMBER
```

Пусть, например, данные будут такими:

```
USERNAME                             DEPTNO
------------------------------ ----------
MARTIN                                 10
MARTIN                                 20
KING                                   20
KING                                   10
KING                                   30
KING                                   40
```

Пользователь Martin может видеть данные об отделах 10 и 20, а пользователь King – 10, 20, 30 и 40. Если имени пользователя в таблице нет, ему не должны быть видны никакие записи. По новым требованиям пользовательские привилегии могут меняться динамически посредством обновления таблицы EMP_ACCESS. Новые привилегии должны вступать в силу сразу, не требуя выхода пользователя из системы и его повторной регистрации.

Новые требования не позволяют полагаться на триггер LOGON при установке значений, используемых функцией политики.

Для соответствия новым условиям можно было бы создать пакет, переменная которого будет хранить предикат, а пользователя снабдить PL/SQL-программой, которая присваивает значение этой переменной. Функция политики тогда могла бы использовать значение, кэшированное в пакете. Допустим ли такой подход? Давайте рассмотрим все внимательно. Если пользователь может изменить значение переменной пакета, что помешает ему присвоить ей значение высокого уровня доступа, как для пользователя King? Martin может войти в систему, задать значение переменной, обеспечивающее ему доступ к сведениям по всем отделам, и осуществить выборку всех записей таблицы. Исчезла конфиденциальность, что неприемлемо. Ведь именно чтобы защититься от подобных действий пользователя, мы обычно помещаем код установки значений, используемых функцией политики, в триггер LOGON.

Возможность динамического изменения пользователем значения переменной пакета требует от нас пересмотра стратегии. Необходимо задавать глобальную переменную каким-то безопасным способом, не допускающим неавторизованного изменения. К счастью, Oracle предоставляет нам такую возможность: следует использовать *контексты приложения*. Контекст приложения является аналогом глобальной переменной пакета: будучи единожды заданным, он доступен на протяжении всего сеанса и может быть задан повторно.

Контекст приложения также напоминает структуру языка C (*struct*) или запись языка PL/SQL. Он состоит из последовательности атрибутов, каждый из которых представляет собой пару имя–значение. Од-

нако, в отличие от своих аналогов в C и PL/SQL, атрибуты не именуются при создании контекста. Они получают имена и значения в процессе выполнения. Контексты приложений хранятся в глобальной области процесса (Process Global Area – PGA).

Механизм задания контекста приложения делает его использование более надежным, чем применение переменной пакета. Изменить значение контекста приложения можно только вызовом специальной программы PL/SQL. Разрешая изменение контекста приложения только специальной процедуре, вы можете обеспечить безопасность, необходимую для реализации политик, значения которых динамически меняются в течение одного сеанса.

Простой пример

Используем команду CREATE CONTEXT для определения нового контекста DEPT_CTX. Любой пользователь, обладающий системной привилегией CREATE ANY CONTEXT и привилегией EXECUTE для пакета DBMS_SESSION, может создавать и настраивать контексты.

```
SQL> CREATE CONTEXT dept_ctx USING set_dept_ctx;
Context created.
```

Обратите внимание на предложение USING set_dept_ctx. Оно указывает на то, что атрибут контекста dept_ctx может задаваться или изменяться только через вызов процедуры set_dept_ctx.

Пока мы еще не задали никаких атрибутов контекста, а только определили его в целом (имя и надежный механизм его изменения). Теперь создадим процедуру. Внутри нее мы будем присваивать значения атрибутам контекста при помощи функции SET_CONTEXT встроенного пакета DBMS_SESSION:

```
CREATE PROCEDURE set_dept_ctx (
   p_attr IN VARCHAR2, p_val IN VARCHAR2)
IS
BEGIN
   DBMS_SESSION.set_context ('DEPT_CTX', p_attr, p_val);
END;
```

Теперь, если мы еще находимся в том же сеансе, которому принадлежит данная процедура, можно вызвать ее напрямую для установки атрибута DEPTNO в значение 10 следующим образом:

```
SQL> EXEC set_dept_ctx ('DEPTNO','10')
PL/SQL procedure successfully completed.
```

Для получения текущего значения атрибута вызываем функцию SYS_CONTEXT, которая принимает два параметра: имя контекста и имя атрибута, например:

```
SQL> DECLARE
  2     l_ret   VARCHAR2 (20);
```

```
  3  BEGIN
  4     l_ret := SYS_CONTEXT ('DEPT_CTX', 'DEPTNO');
  5     DBMS_OUTPUT.put_line ('Value of DEPTNO = ' || l_ret);
  6  END;
     /

Value of DEPTNO = 10
```

Возможно, вы помните, что функция SYS_CONTEXT уже использовалась в этой главе для получения IP-адреса и имени терминала клиента.

Безопасность контекстов приложения

Процедура set_dept_ctx фактически инкапсулирует вызов функции SET_CONTEXT с определенными параметрами. Почему бы не вызывать встроенную функцию напрямую? Давайте посмотрим, что произойдет, если пользователь вызовет тот же самый фрагмент кода для установки значения атрибута DEPTNO в 10.

```
SQL> BEGIN
  2     DBMS_SESSION.set_context ('DEPT_CTX', 'DEPTNO', 10);
  3  END;
  4  /
begin
*
ERROR at line 1:
ORA-01031: insufficient privileges
ORA-06512: at "SYS.DBMS_SESSION", line 82
ORA-06512: at line 2
```

Обратите внимание на сообщение об ошибке: «ORA-01031: insufficient privileges» (недостаточно привилегий). Оно может привести в замешательство, так как пользователь как раз *обладает* необходимой привилегией EXECUTE на пакет DBMS_SESSION (без этой привилегии не удалось бы скомпилировать set_dept_ctx).

Недостаточность привилегий относится не к использованию DBMS_SESSION, а к попытке задания значения контекста вне процедуры set_dept_ctx.

Как видите, Oracle «доверяет» задание значений контекста приложения только процедуре set_dept_ctx. В Oracle процедура, которая указана в предложении USING оператора CREATE CONTEXT, называется *доверенной* (*trusted*).

Выполнять доверенную процедуру могут только следующие схемы:

- Схема, которой принадлежит данная процедура.
- Любая схема, которой выдана привилегия EXECUTE для данной доверенной процедуры.

Таким образом, аккуратное распределение привилегий EXECUTE может обеспечить полный контроль над заданием значений данного контекста.

Доверенная процедура должна быть указана в момент создания контекста приложения. Только доверенная процедура сможет устанавливать значения контекста.

Контексты как предикаты RLS

Мы узнали о том, что для задания значения контекста должна использоваться процедура, а это аналогично использованию глобальной переменной пакета. У вас может возникнуть вопрос о разумности такого подхода: не создаем ли мы дополнительные сложности, ничего определенного при этом не добиваясь.

Нет. Доверенная процедура – это единственное средство задания значения атрибута контекста, и потому ее можно использовать для контроля выполнения. Внутрь доверенной процедуры можно поместить любые проверки корректности присвоения значения переменной. Можно даже полностью устранить передачу параметров и устанавливать значения из предопределенных значений, без ввода (и соответственно воздействия) со стороны пользователя. Например, возвращаясь к требованиям по управлению доступом к данным о сотрудниках, список номеров отделов для передачи в контекст приложения можно получать из таблицы EMP_ACCESS, а не от пользователя.

Будем использовать контекст приложения внутри самой функции политики. Начнем с изменения функции политики.

```
 1 CREATE OR REPLACE FUNCTION authorized_emps (
 2     p_schema_name   IN   VARCHAR2,
 3     p_object_name   IN   VARCHAR2
 4  )
 5     RETURN VARCHAR2
 6  IS
 7     l_deptno       NUMBER;
 8     l_return_val   VARCHAR2 (2000);
 9  BEGIN
10     IF (p_schema_name = USER)
11     THEN
12        l_return_val := NULL;
13     ELSE
14        l_return_val := SYS_CONTEXT ('DEPT_CTX', 'DEPTNO_LIST');
15     END IF;
16
17     RETURN l_return_val;
18  END;
```

Функция политики ожидает передачи номеров отделов от атрибута DEPTNO_LIST контекста DEPT_CTX (строка 14). Для задания этого значения необходимо изменить доверенную процедуру контекста:

```
CREATE OR REPLACE PROCEDURE set_dept_ctx
IS
```

```
   l_str   VARCHAR2 (32767);
   l_ret   VARCHAR2 (32767);
BEGIN
   FOR deptrec IN (SELECT deptno
                     FROM emp_access
                    WHERE username = USER)
   LOOP
      l_str := l_str || deptrec.deptno || ',';
   END LOOP;

   IF l_str IS NULL
   THEN
      -- нет данных о доступе, ничего не выводим.
      l_ret := '1=2';
   ELSE
      l_ret := 'DEPTNO IN (' || RTRIM (l_str, ',') || ')';
      DBMS_SESSION.set_context ('DEPT_CTX', 'DEPTNO_LIST', l_ret);
   END IF;
END;
```

Давайте протестируем функцию. Сначала пользователь Martin входит в систему и вычисляет количество сотрудников. Перед выдачей запроса ему необходимо задать контекст.

```
SQL> EXEC rlsowner.set_dept_ctx
PL/SQL procedure successfully completed.

SQL> SELECT sys_context ('DEPT_CTX','DEPTNO_LIST') FROM dual;

SYS_CONTEXT('DEPT_CTX','DEPTNO_LIST')
-------------------------------------

DEPTNO IN (20,10)

SQL> SELECT DISTINCT deptno FROM hr.emp;

    DEPTNO
----------
        10
        20
```

Martin видит только данные сотрудников отделов 10 и 20, как и предусмотрено таблицей EMP_ACCESS.

Предположим теперь, что права доступа Martin изменены: теперь ему будут доступны записи об отделе 30, для чего выполнены соответствующие изменения в таблице EMP_ACCESS:

```
SQL> DELETE emp_access WHERE username = 'MARTIN';
2 rows deleted.

SQL> INSERT INTO emp_access values ('MARTIN',30);
1 row created.

SQL> COMMIT;
Commit complete.
```

Когда Martin попытается выполнить тот же запрос, что и раньше, он получит другие результаты. Сначала выполняется хранимая процедура, задающая атрибут контекста.

```
SQL> EXEC rlsowner.set_dept_ctx
PL/SQL procedure successfully completed.

SQL> SELECT sys_context ('DEPT_CTX','DEPTNO_LIST') FROM dual;

SYS_CONTEXT('DEPT_CTX','DEPTNO_LIST')
--------------------------------------------------------------

DEPTNO IN (30)

SQL> SELECT DISTINCT deptno FROM hr.emp;

    DEPTNO
----------
        30
```

Изменения вступают в силу автоматически. Как видите, Martin не указывает, какие отделы ему разрешено видеть, а просто вызывает хранимую процедуру `set_dept_ctx`, которая автоматически задает атрибуты контекста. Пользователь не может самостоятельно задать атрибуты контекста, что делает данный метод более надежным, чем использование глобальной переменной пакета (которую Martin мог бы напрямую установить в любое значение).

Что будет, если Martin не выполнит процедуру `set_dept_ctx` перед выдачей запроса `SELECT`? На момент выполнения запроса атрибут `DEPTNO_LIST` контекста приложения `DEPT_CTX` будет содержать значение `NULL`, следовательно, предикат политики не будет включать в себя ни одного номера отдела. В результате Martin не сможет видеть данные ни об одном сотруднике.

Давайте внимательно проанализируем ситуацию. Мы создали предикат политики (другими словами, условие `WHERE`), который должен применяться к пользовательскому запросу. Мы решили, что будем сначала задавать атрибут контекста приложения, а функция политики будет обращаться к атрибуту контекста, а не к таблице `EMP_ACCESS`. Можно было бы сделать и так, чтобы функция политики обращалась напрямую к таблице `EMP_ACCESS` и создавала предикат: это значительно упростило бы написание функции политики. В этом случае пользователю не пришлось бы выполнять функцию политики при каждом входе в систему.

Однако функция политики, осуществляющая выборку из контекста приложения, а не напрямую из таблицы, имеет свои преимущества. Давайте сравним два подхода, используя псевдокод для представления базовой логики.

Сначала поместим все необходимые действия в функцию политики: осуществляем выборку из таблицы `EMP_ACCESS` и возвращаем строку предиката.

```
1  Получить имя пользователя
2  Цикл
3        Выбрать из таблицы EMP_ACCESS номера отделов,
4        которые доступны для данного имени пользователя
5        Составить список номеров отделов
6  Конец цикла
7  Вернуть список в качестве предиката
```

Теперь сделаем то же самое в процедуре `set_dept_ctx`:

```
1  Получить имя пользователя
2  Цикл
3        Выбрать из таблицы EMP_ACCESS номера отделов,
4        которые доступны для данного имени пользователя
5        Составить список номеров отделов
6  Конец цикла
7  Установить атрибут DEPTNO_LIST в значение полученного списка
```

Тогда в функции политики будет выполняться только следующее:

```
1  Найти атрибут контекста DEPTNO_LIST
2  Вернуть его в качестве предиката политики
```

Обратите внимание на различия двух подходов. После входа пользователя в систему его имя в течение сеанса не изменяется. Поэтому функция `set_dept_ctx` может быть выполнена единожды – при начале сеанса, для задания атрибута контекста. Функция политики, созданная вокруг этого атрибута контекста, тем самым избегает обращения к базовой таблице `EMP_ACCESS` и полагается исключительно на память сеанса.

Если использовать ту версию функции политики, которая осуществляет выборку из таблицы, то операторам SQL, запускающим функцию политики, придется делать гораздо больше работы. То есть политики, которые обращаются ко всем необходимым данным через контексты приложения, могут значительно улучшить производительность операторов SQL, на которые наложены политики RLS.

В Oracle 10*g* использование контекстов имеет дополнительное преимущество. Вы можете определить политику как контекстно-зависимую (см. раздел «RLS в Oracle 10*g*»), – это означает, что функция политики будет выполняться только при изменении контекста. Для нашего примера, в таком случае, функция политики будет выполнена всего один раз (когда пользователь входит в систему и задает контекст), поэтому политика будет применяться очень быстро. При изменении условий предоставления доступа пользователь повторно выполняет процедуру `set_dept_ctx`, которая повторно выполнит функцию политики.

Идентификация пользователей, не зарегистрированных в базе данных

Контексты приложения могут использоваться далеко за пределами тех ситуаций, которые были нами рассмотрены. Основное предназна-

чение контекстов приложения в том, чтобы различать пользователей, которых невозможно идентифицировать на основе уникальности сеансов. Веб-приложения регулярно используют *пул соединений* с базой данных, когда соединение осуществляется через некоторого пользователя, например CONNPOOL. Веб-пользователи подключаются к серверу приложений, который в свою очередь использует одно из соединений пула для обращения к базе данных (рис. 5.2).

Пользователи Martin и King не являются пользователями базы данных. Это веб-пользователи, и база данных не обладает никакими специфическими сведениями о них. Пул соединений подключается к базе данных через пользователя с идентификатором CONNPOOL, который зарегистрирован в базе данных. Когда Martin запрашивает что-то из базы данных, пул может решить использовать соединение, помеченное номером 1, для получения данных. После выполнения запроса соединение переходит в режим ожидания. Если в этот момент пользователь King захочет выполнить запрос, пул вполне может решить использовать то же самое соединение (помеченное 1). С точки зрения базы данных сеанс (который на самом деле является соединением из пула) относится к пользователю CONNPOOL. Поэтому в рассмотренном ранее примере (где функция USER идентифицирует реальное имя схемы) невозможно будет добиться уникальной идентификации пользователя, выполняющего вызовы. Функция USER будет всегда возвращать CONNPOOL, так как к базе данных подключен именно этот пользователь.

Тут в дело вступает контекст приложения. Предположим, что существует контекст WEB_CTX с атрибутом WEBUSER. При получении запроса от клиента это значение устанавливается пулом соединений в имя реального пользователя (например, Martin). Политика RLS может основываться на данном значении, а не на имени пользователя базы данных.

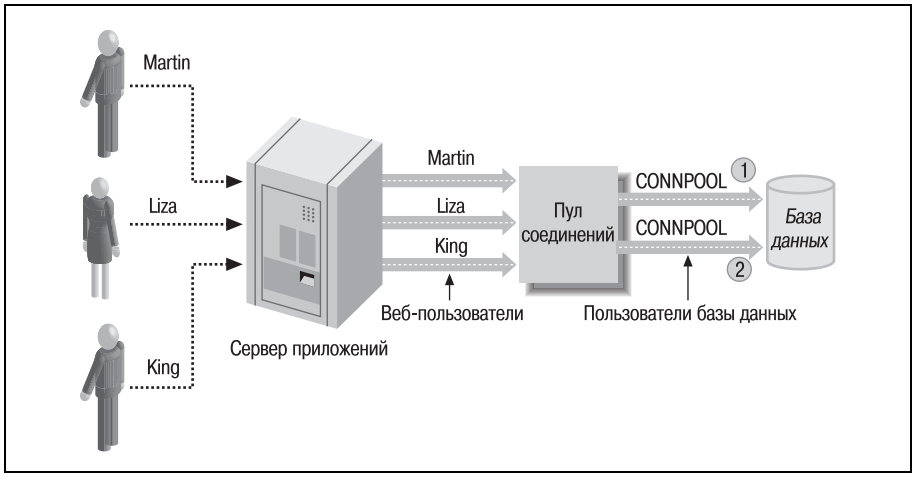


Рис. 5.2. *Пользователи приложений и RLS*

Посмотрим, как это будет работать. Пусть у нас есть банковское приложение, в котором доступ к записям клиентов осуществляют несколько менеджеров. Необходимо построить политику RLS так, чтобы каждый менеджер видел счета только собственных клиентов. Столбец ACC_MGR хранит имя пользователя для менеджера, работающего с соответствующим счетом. Тогда предикат политики может быть таким:

```
ACC_MGR = AccountManagerUserName
```

где *AccountManagerUserName* – это идентификатор пользователя Windows-менеджера (то есть информация, не известная базе данных). Данное значение должно быть передано пулом соединений базе данных посредством контекстов.

Начнем с создания контекста:

```
CREATE CONTEXT web_ctx USING set_web_ctx;
```

Основная процедура, задающая контекст, будет выглядеть следующим образом:

```
CREATE OR REPLACE PROCEDURE set_web_ctx (p_webuser IN VARCHAR2)
IS
BEGIN
   DBMS_SESSION.set_context ('WEB_CTX', 'WEBUSER', p_webuser);
END;
```

Процедура принимает один параметр, имя реального пользователя (веб-пользователя). Именно эти данные будут использованы приложением для задания контекста WEB_CTX. Проверим, что процедура работает:

```
SQL> EXEC set_web_ctx ('LIZA')

PL/SQL procedure successfully completed.

SQL> EXEC DBMS_OUTPUT.put_line(sys_context('WEB_CTX','WEBUSER'))

LIZA

PL/SQL procedure successfully completed.
```

Приведенная процедура задания контекста (set_web_ctx) очень проста. Она лишь задает атрибут контекста. В реальной жизни вам придется писать множество строк кода, выполняющих различные проверки на наличие у вызывающего соответствующих прав, и т. д. Например, сервер приложений под управлением Windows может напрямую извлекать имя пользователя с клиентского компьютера и передавать его в контекст, используя описанную выше процедуру.

Задав контекст, используем его для построения функции политики:

```
CREATE OR REPLACE FUNCTION authorized_accounts (
   p_schema_name   IN   VARCHAR2,
   p_object_name   IN   VARCHAR2
)
```

```
   RETURN VARCHAR2
IS
   l_deptno       NUMBER;
   l_return_val   VARCHAR2 (2000);
BEGIN
   IF (p_schema_name = USER)
   THEN
      l_return_val := NULL;
   ELSE
      l_return_val :=
                  'acc_mgr = ''' || SYS_CONTEXT ('WEB_CTX', 'WEBUSER')
                  || '''';
   END IF;

   RETURN l_return_val;
END;
```

Функция политики возвращает предикат `acc_mgr = '`*имя_пользователя*`'`, который применяется к пользовательским запросам. Пользователь автоматически получает доступ только к собственным записям.

Заключение

Средства RLS чрезвычайно важны для обеспечения безопасности базы данных на уровне строк. Область применения RLS (учитывая всю полезность этой технологии для требующих защиты приложений и баз данных) выходит за рамки обеспечения безопасности. Средства RLS могут использоваться для ограничения доступа к некоторым строкам таблицы, избавляя от необходимости модифицировать приложения при изменении условий запроса; возможен также выборочный перевод таблиц в режим доступа только для чтения. Используя комбинации переменных внутри функции политики, вы можете создать настраиваемое представление данных внутри таблицы, что позволит удовлетворить потребности пользователей и создать удобное для поддержки приложение.

6

Детальный аудит

Аудитом называется механизм регистрации действий пользователей базы данных. Позволяя определить, кем из пользователей выполнены определенные действия, аудит обеспечивает *контролируемость* (*accountability*), которая является краеугольным камнем безопасности. Традиционный аудит в Oracle сохраняет сведения о сделанных пользователями изменениях данных, но не о выполненных ими запросах. *Детальный аудит* (*Fine-grained audit, FGA*), введенный в Oracle9*i*, расширяет возможности регистрации, позволяя сохранять сведения как об изменениях, так и о запросах. Детальный аудит очень важен с точки зрения безопасности; помимо этого он служит прекрасным средством анализа использования SQL и оценки производительности как отдельных операторов, так и приложения в целом. Он предоставляет метод для исследования типовых обращений к данным, который может стать мощным инструментом для повышения производительности базы данных.

Эта глава поможет вам в эффективном использовании детального аудита, который (позволяя выбирать, какие события и какие сведения о них должны регистрироваться) может настраиваться для наилучшего соответствия требованиям вашей базы данных и приложений.

Функциональность детального аудита Oracle базируется на встроенном пакете `DBMS_FGA`. В этой главе описаны программы `DBMS_FGA`, позволяющие создавать и использовать политики детального аудита; также рассмотрено соответствие возможностей FGA в версиях Oracle9*i* и Oracle 10*g*. Кроме того, рассматривается взаимодействие детального аудита с другим нововведением Oracle 10*g* – *ретроспективными запросами* (*flashback query*), позволяющими точно определить, какие результаты получали пользователи при выполнении своих запросов (а не то, что в данный момент находится в базе данных). Затем мы сравним FGA с триггерами базы данных, которые традиционно использовались для реализации отдельных функций детального аудита.

В силу того, что многие администраторы еще работают с Oracle9*i*, эта глава начинается с описания возможностей FGA этой версии, большая часть которых аналогично реализуется в Oracle 10*g*. Затем в разделе «FGA в Oracle 10*g*» описываются усовершенствования, сделанные в Oracle 10*g*.

Не путайте аббревиатуры FGA и FGAC; последняя означает «детальный контроль доступа» (*fine-grained access control*) – другое название технологии RLS, описанной в главе 5.

Введение в детальный аудит

Чтобы извлечь максимум возможного из технологии FGA, начнем с изучения общих требований к аудиту, а затем рассмотрим простой пример использования детального аудита FGA.

Для обеспечения корректной работы FGA необходимо, чтобы база данных функционировала в режиме оптимизации по стоимости. Запросы должны использовать оптимизатор по стоимости (то есть они не должны использовать хинты `RULE`); таблицы (или представления) запроса должны анализироваться хотя бы на основе приблизительных оценок. В противном случае возможно ложное срабатывание FGA, то есть запись в журнал аудита будет вестись даже в отсутствии необходимости. Имейте в виду, что Oracle 10*g* использует оптимизатор по стоимости по умолчанию, о чем мы поговорим далее.

Что такое аудит?

Одним из основных требований компьютерной безопасности является *контролируемость*, то есть возможность отслеживания действий пользователя таким образом, чтобы впоследствии можно было сопоставить действия выполнившим их пользователям. Oracle обеспечивает контролируемость за счет применения аудита: отслеживания того, кто что сделал. Предположим, что пользователь Scott выполняет такой запрос:

```
SELECT * FROM emp;
```

Если аудит корректно настроен, то база данных зарегистрирует факт того, что пользователь Scott что-то выбрал из таблицы `EMP`, а также запишет ряд других данных: время доступа, использованный терминал и т. д. Записываемая информация называется *журналом аудита* (*audit trail*) и при необходимости может использоваться для последующего анализа действий Scott. Журнал аудита принадлежит пользователю `SYS`, поэтому обычные пользователи не могут изменить его содержимое и соответственно скрыть сведения о своих действиях. Традиционный журнал аудита Oracle хранится в таблице `AUD$`, принадлежащей пользователю `SYS`. Информация журнала доступна пользователям через такие

представления словаря данных, как DBA_AUDIT_TRAIL. Журнал аудита может храниться не только в таблицах базы данных, но и в файлах операционной системы.

По умолчанию действия пользователя SYS не отслеживаются. Для того чтобы выполнять аудит и для его объектов, следует установить параметр инициализации базы данных AUDIT_SYS_OPERATIONS в значение TRUE. Однако в этом случае записи аудита будут вноситься в файлы операционной системы, а не в таблицы базы данных. Общее эмпирическое правило говорит о том, что никогда не следует выполнять обычные операции DML от имени SYS.

К сожалению, обычный аудит имеет серьезные ограничения: записывается лишь сам факт того, что Scott что-то выбрал из таблицы EMP, но не результат выполнения запроса. А чаще всего вопрос «Что?» не менее важен, чем «Кто?». Например, в финансовых компаниях финансовые данные клиентов должны быть защищены от любопытных. Для обеспечения конфиденциальности компания может захотеть регистрировать пользователей, просматривающих секретные клиентские данные. В этом случае необходимо записывать не только личность пользователя, обращавшегося к данным, но и то, какие именно данные просматривались. Обычный аудит не позволяет определить, какие записи просматривали пользователи, фиксируется только факт просмотра каких-то данных в таблице.

Один из способов сбора информации об изменяющих данные операторах DML заключается в создании триггеров для таблицы. Однако обычно администратор базы данных хочет знать не только об изменении данных, но и о выборке. Триггеры не могут быть сопоставлены операторам SELECT, поэтому получить подобную информацию из DML-триггеров для строк невозможно. В версиях, предшествующих Oracle9*i*, вообще не было возможности сбора сведений об операторах SELECT.

Начиная с Oracle9*i* эту брешь заполняет FGA. Используя эту технологию, вы можете отслеживать действия над таблицами, не создавая (и не поддерживая) триггеры. Oracle записывает сведения FGA в отдельную таблицу (не в ту, которая используется для обычного аудита) – FGA_LOG$ в схеме SYS. Этой таблице соответствует представление словаря данных DBA_FGA_AUDIT_TRAIL.

Далее в этой главе мы будем обсуждать и сравнивать различные виды аудита, поддерживаемые Oracle в настоящее время. Вы сможете выбрать тот подход, который наиболее полно отвечает требованиям вашего приложения.

Зачем нам знать об FGA?

Существует ряд причин, по которым FGA является чрезвычайно полезным инструментом для администратора базы данных. Сейчас мы познакомим вас с ними, а затем изучим эти вопросы подробно.

Повышение безопасности

Конечно же, основным предназначением FGA является обеспечение безопасности. Возможность регистрации действий пользователя, производимых над базой данных, чрезвычайно важна для безопасности, а FGA предоставляет наилучший способ записи сведений о подобных действиях (в некоторых случаях это и единственно возможный способ, так как традиционный аудит не позволяет собирать данные о том, какой именно запрос был выдан пользователем).

Анализ выполнения SQL

FGA выявляет, кто что сделал. Видя реальные SQL-операторы, выполненные пользователями, администратор базы данных может судить о типах операторов, которые выдаются из приложений конкретными пользователями в определенные моменты времени. Такая информация полезна при принятии решения об индексировании схемы или при анализе повторяемости. Она еще более полезна для систем поддержки принятия решений (Decision Support Systems), в которых запросы обычно специально подбираются к каждому случаю и не могут быть предсказаны заранее. Принимая во внимание возможность отображения другой важной информации, такой как временная метка и имя терминала, а также то, что вся информация находится в таблице, диагностика проблем и анализ доступа значительно упрощаются.

Оптимизация производительности при использовании переменных связывания

Журнал аудита FGA также содержит информацию о значениях переменных связывания, которые широко используются в любом хорошо спроектированном приложении. Откуда можно было бы узнать о том, какие различные значения передаются в ходе работы приложения? Такая информация помогла бы упростить и ускорить принятие решения о необходимости создания индекса для таблицы. Ее предоставит вам журнал аудита FGA.

Эмуляция триггеров SELECT

Хотя такая возможность и не проектировалась специально, скрытая ценность FGA заключается в возможности определения или эмуляции «триггеров на SELECT», которые никаким другим способом в Oracle не доступны. FGA позволяет указать необходимость выполнения PL/SQL-процедуры при выборке данных, так же, как и для изменения данных посредством операторов `INSERT`, `UPDATE` или `DELETE`. FGA имитирует триггер на `SELECT`, автоматически выполняя хранимую процедуру при выдаче пользователем запроса. Вы можете и далее контролировать выполнение запроса, применяя различные условия (например, выбираемые столбцы, значения использованных столбцов).

Простой пример

Давайте рассмотрим простой пример использования FGA, иллюстрирующий основные возможности детального аудита. Предположим, что таблица EMP в схеме HR базы данных отдела кадров определена следующим образом:

```
SQL> DESC emp
 Name              Null?    Type
 ----------------- -------- ------------
 EMPID             NOT NULL NUMBER(4)
 EMPNAME                    VARCHAR2(10)
 JOB                        VARCHAR2(9)
 MGR                        NUMBER(4)
 HIREDATE                   DATE
 SALARY                     NUMBER(7,2)
 COMM                       NUMBER(7,2)
 DEPTNO                     NUMBER(2)
```

Для удовлетворения требований безопасности и конфиденциальности необходимо отслеживать любые запросы к этой таблице. Использование триггеров или традиционного аудита не подходит, так что обратимся к FGA. Начинаем с создания *политик* посредством программы ADD_POLICY пакета DBMS_FGA:

```
BEGIN
   DBMS_FGA.add_policy (object_schema      => 'HR',
                        object_name        => 'EMP',
                        policy_name        => 'EMP_SEL'
                       );
END;
```

Для выполнения кода, приведенного в примере, пользователь должен обладать привилегией EXECUTE на пакет DBMS_FGA или запускать код от имени пользователя SYS.

В целом политика FGA управляет отслеживанием операторов, выполняемых для определенной таблицы. Она определяет условия, при которых включается аудит, а также предпринимаемые действия. В предыдущем примере политика EMP_SEL создается и применяется к таблице EMP схемы HR. Имя политики должно быть уникальным в рамках экземпляра базы данных. Для одной таблицы можно определить до 256 отдельных политик. Использование нескольких политик позволяет обеспечить большую гибкость и упростить управление ими. Мы подробнее поговорим об определении политик и управлении ими ниже в этой главе.

Термин «политика», используемый как в рамках FGA, так и RLS, понимается при этом по-разному (о чем уже говорилось в этой главе и в главе 5). Политика FGA похожа на свою RLS-«тезку» тем, что она не является объектом схемы (то есть не принадлежит никакому пользователю). Любой пользователь с при-

> вилегией EXECUTE на пакет DBMS_FGA может создать и удалить политику, созданную другим пользователем. Поэтому разумно отзывать привилегию EXECUTE для роли PUBLIC (если она была выдана). Следует выдавать привилегию EXECUTE на данный пакет только тем пользователям, которым она действительно необходима.

Политика добавлена – теперь таблица находится под надзором, и действия любого обращающегося к ней пользователя будут регистрироваться. Так что если пользователь Scott выполнит такой оператор:

```
SELECT salary
  FROM hr.emp
 WHERE empid = 1;
```

то он будет записан в журнал аудита SYS.FGA_LOG$.

Для просмотра журнала аудита выполните запрос к представлению словаря данных DBA_FGA_AUDIT_TRAIL. (Естественно, для этого необходимо обладать привилегией SELECT на это представление, которая может быть получена через роль или быть выдана напрямую, как SELECT ANY DICTIONARY.) Например:

```
SELECT db_user, sql_text
  FROM dba_fga_audit_trail
 WHERE object_schema = 'HR' AND object_name = 'EMP';
```

Для запроса, выполненного только что к таблице EMP, выводятся следующие данные:

```
DB_USER SQL_TEXT
------- ------------------------------------------------
SCOTT   select salary from hr.emp where empid = 1
```

Зафиксирован не только сам факт выборки данных из таблицы пользователем Scott, но и точный текст выполненного запроса.

Если для таблицы определена политика FGA, то такую таблицу нельзя преобразовать средствами встроенного пакета Oracle DBMS_REDEFINITION. Попытка переопределения таблицы, находящейся под контролем FGA, приводит к ошибке ORA-12090: «cannot online redefine table». Поэтому перед созданием политики FGA для таблицы следует подумать о возможности ее реорганизации в будущем.

Аудит и различные версии СУБД Oracle

Аудит и детальный аудит отличаются для версий Oracle9*i* и Oracle 10*g*. Здесь перечислены основные различия. Более подробная информация об использовании этих особенностей будет приведена ниже (в частности, в разделе «FGA в Oracle 10*g*»).

- В Oracle 10*g* расширена функциональность оператора AUDIT, используемого в обычном аудите для регистрации факта выборки из опре-

деленной таблицы. Теперь он может собирать информацию о выполненном операторе SQL, что было невозможно в предыдущей версии. (Может показаться, что усовершенствование делает обычный и детальный аудит в версии Oracle 10*g* практически идентичными, но на самом деле это не так, о чем еще будет рассказано дальше.)

- В Oracle9*i* детальный аудит ведется только для операторов `SELECT`, но не для таких операторов DML, как `INSERT`, `UPDATE` и `DELETE`. В Oracle9*i* собрать информацию о том, что было изменено, можно только посредством создания триггеров для этих операторов и внесения записей в журнальную таблицу. В Oracle 10*g* механизм FGA позволяет собирать сведения и об этих операторах DML. Хотя триггеры не очень часто востребованы при работе с Oracle 10*g*, но иногда имеет смысл использовать именно триггеры, а не FGA. Оба подхода имеют свои достоинства и недостатки, которые будут рассмотрены ниже.
- В Oracle9*i* детальный аудит запускается при ссылке на любой столбец из списка, заданного при создании политики. В Oracle 10*g* предоставлена возможность выбора: выполнять аудит при ссылке на любой столбец из списка или только при упоминании всех столбцов списка.

Какие еще сведения собирает FGA?

В рассмотренном ранее примере использования аудита мы выбирали данные о пользователе, выполнившем запрос, и о тексте запроса. В журнал аудита записывается еще много другой информации, при этом важнейшим является время выполнения действия. Столбец `TIMESTAMP` представления `DBA_FGA_AUDIT_TRAIL` хранит временную метку, для просмотра которой вы, вероятно, захотите использовать следующий формат (для отображения полного значения):

```
TO_CHAR(TIMESTAMP,'mm/dd/yyyy hh24:mi:ss')
```

Еще ряд полезных столбцов позволяет установить личность пользователя и получить подробные сведения об отслеживаемых действиях (что поможет обеспечить контролируемость и удобство анализа). Перечислим наиболее важные столбцы представления `DBA_FGA_AUDIT_TRAIL`:

`DB_USER`

Пользователь, выполнивший оператор.

`SQL_TEXT`

Текст выполненного пользователем SQL-оператора.

`TIMESTAMP`

Момент времени, в который пользователь выполнил действие.

`OS_USER`

Имя пользователя операционной системы, подключившегося к базе данных.

USERHOST

Терминал или клиентский компьютер, с которого было осуществлено подключение.

EXT_NAME

Пользователь может проходить внешнюю аутентификацию (например, через LDAP). В этом случае имя пользователя в системе внешней аутентификации является важным параметром, который записывается в данный столбец.

SQL_BIND

Значения переменных связывания, использованных в запросе (при их наличии).

FGA и ретроспективные запросы

Для того чтобы понять, почему полезно использовать детальный аудит в сочетании с ретроспективными запросами Oracle, рассмотрим один пример. Предположим, что я (администратор базы данных) смотрю в журнал аудита и обнаруживаю там, что пользователь Scott выдал такую команду:

```
SELECT salary FROM hr.emp WHERE empid = 100;
```

Так случилось, что служащий с номером 100, – это я, поэтому я испытываю шок от того, что Scott подглядел, какая у меня зарплата. Он уже нацеливался на мое место, так что неудивительно, что он решил проверить мою зарплату. Потом мне в голову приходит такая мысль: за последнее время у меня было несколько повышений, изменивших мою зарплату с 12000 до 13000, затем до 14000 и наконец до 15000. Интересно, что увидел Scott: новую или старую зарплату. И если старую, то какую именно: 12000, 13000 или 14000? Если я сам выполню тот же запрос, что и Scott, то увижу текущую величину – 15000, а не то старое значение, которое было доступно при выдаче запроса Scott.

Oracle9*i* поддерживает замечательную функциональность, называемую *ретроспективным запросом* (*flashback query*), которая поможет ответить на наши вопросы. С ее помощью можно выбирать значение на определенный момент времени в прошлом вне зависимости от того, изменялось ли оно и/или фиксировалось впоследствии. Давайте посмотрим, как это работает.

Предположим, что изначально величина зарплаты равнялась 12000.

```
SQL> SELECT salary FROM emp WHERE empid = 100;

    SALARY
----------
     12000
```

Затем 10 июля она была повышена моим начальником до 13000, информация о чем была занесена в базу данных.

```
SQL> UPDATE emp set salary = 13000 WHERE empid = 100;

1 row updated.

SQL> COMMIT;
```

Однако такое повышение меня не устроило. 11 июня после повторных переговоров моя зарплата была увеличена до 14000.

```
SQL> UPDATE emp set salary = 14000 WHERE empid = 100;

1 row updated.

SQL> COMMIT;
```

Но я все еще хотел большего. (Ну что я могу сказать? У меня очень много расходов.) После долгих дискуссий мне удалось убедить моего начальника в том, что такие работники, как я, на вес золота. 12 июня значение было изменено на 15000.

```
SQL> UPDATE emp set salary = 15000 WHERE empid = 100;

1 row updated.

SQL> COMMIT;
```

Сегодня, 13 июня, моя зарплата составляет 15000. Для того чтобы увидеть ее величину на 9 июня, следует использовать ретроспективный запрос. Синтаксис AS OF позволяет создать такой запрос:

```
SQL> SELECT salary FROM emp AS OF TIMESTAMP
  2  TO_TIMESTAMP('6/9/2004 01:00:00','MM/DD/YYYY HH24:MI:SS')
  3  WHERE empid = 100;

    SALARY
----------
     12000
```

Точно так же, указывая другие временные метки, можно выяснить значения столбца в другие моменты времени.

Если вы указываете временную метку в предложении AS OF после имени таблицы, то Oracle получает значение из сегментов отката, а не из реальной таблицы (при условии, что сегмент отката достаточно велик и в нем содержится предшествующее изменению значение столбца). В рассмотренном примере предполагается, что сегмент отката хранит изменения за четыре дня. Определить или изменить это значение можно через параметр инициализации базы данных UNDO_RETENTION (задается в секундах). Выбранное мною значение в 4 дня вполне обычно для медленно изменяющихся баз данных (как наша HR), но в быстро меняющихся OLTP базах данных (например, в системах бронирования) следует выбрать большее значение.

При внесении изменения в базу данных Oracle записывает возрастающий счетчик – *системный номер изменения* (System Change Number – SCN), который однозначно идентифицирует изменения. Благодаря то-

му, что каждому изменению сопоставлен номер SCN, Oracle может получить старое значение, найдя соответствующий номер SCN и затем запросив значение для данного номера из сегмента отката.

В предыдущем примере использовалась временная метка, а не номер SCN. Как же Oracle сопоставляет номер SCN временной метке, и наоборот? Для этого используется таблица SMON_SCN_TIME, поддерживаемая процессом SMON, который записывает временную метку и соответствующий ей номер SCN. Следует, однако, иметь в виду, что номера SCN записываются с интервалом в пять минут. Приведем пример использования таблицы:

```
SELECT time_dp, scn_bas FROM sys.smon_scn_time
```

Вывод будет таким:

```
TIME_DP                SCN_BAS
------------------- ----------
06/26/2004 15:29:26 1167826228
06/26/2004 15:34:33 1167826655
06/26/2004 15:39:41 1167827058
06/26/2004 15:44:48 1167827476
... и так далее ...
```

Обратите внимание на пятиминутные промежутки между временными метками. Если в предложении AS OF ретроспективного запроса указана временная метка, то Oracle считает, что номер SCN остается неизменным на протяжении пяти минут. Например, в приведенном выше выводе отображается номер SCN 1167826228 в 15:29:26, и 1167826655 в 15:34:33, что подразумевает, что пять минут между этими временными метками значение SCN всегда оставалось равным 1167826228. Конечно, это неверно. Номер SCN увеличился с 1167826228 до 1167826655, на 427 единиц за 5 минут, что является результатом последовательных приращений в соответствии с многочисленными изменениями, не отраженными в таблице SMON_SCN_TIME. Ретроспективный запрос не может получить более детальную информацию, чем номер SCN, поэтому он считает, что этот номер остается неизмененным на протяжении пятиминутного интервала. Поэтому запрос ищет один и тот же номер SCN – 1167826228 – в 15:29:26, в 15:30:00 и так вплоть до 15:34:33. Так что, указав любую из этих временных меток в предложении AS OF ретроспективного запроса, вы получите одни и те же данные, потому что Oracle будет производить поиск по одному и тому же номеру SCN.

Использование временной метки в ретроспективном запросе позволяет получить результаты с дискретностью в пять минут, но не более точно. Для получения данных на точный момент времени следует использовать предложение AS OF SCN.

Для получения фактических значений необходимо указывать конкретный номер SCN. Предыдущий запрос можно изменить следующим образом:

```
SELECT salary
  FROM emp AS OF SCN 1167826230
 WHERE empid = 100;
```

Значение будет получено для SCN 1167826230 и в результате будет возвращено точное значение для указанного номера SCN, а не округленное в рамках пятиминутного интервала.

Номер SCN является ключевой составляющей журнала детального аудита. Номера SCN позволяют «вернуться в прошлое» для определения значений, впоследствии подвергшихся изменениям, на определенный момент времени. Столбец SCN представления DBA_FGA_AUDIT_TRAIL регистрирует номера SCN на всем протяжении ведения журнала. Для того чтобы определить, какое именно значение столбца SALARY увидел пользователь Scott, можно выполнить следующий запрос, который получает номер SCN при выборке записи пользователем Scott:

```
SELECT SCN
  FROM dba_fga_audit_trail
 WHERE object_schema = 'HR' AND object_name = 'EMP';
```

Предположим, что запрос вернул 14122310350. Тогда выполняем еще один запрос, теперь уже чтобы узнать точное значение столбца SALARY, которое увидел Scott.

```
SELECT salary
  FROM emp AS OF SCN 14122310350
 WHERE empid = 100;
```

Сведения об SCN в журнале детального аудита чрезвычайно важны, так как сохранение данных, которые пользователь увидел в результате выполнения своего запроса, обеспечивает контролируемость. Помните о том, что объем доступных данных отката зависит от размера табличного пространства отката и значения параметра инициализации базы данных UNDO_RETENTION_PERIOD. Эффективными будут только оперативно выполненные ретроспективные запросы. В противном случае данные могут быть перезаписаны, и получение точных ретроспективных данных окажется невозможным.

При необходимости как можно более точно определить значения столбца на указанный момент времени следует использовать в предложении AS OF не временную метку, а номер SCN.

Настройка FGA

Посмотрим код, использованный в предыдущем разделе, для ввода в действие механизма детального аудита для таблицы:

```
BEGIN
   DBMS_FGA.add_policy (object_schema      => 'HR',
                        object_name        => 'EMP',
                        policy_name        => 'EMP_SEL'
```

```
                        );
END;
```

В примере приведена абсолютно элементарная политика. В реальной жизни вам нужно будет настроить детальный аудит так, чтобы он отвечал вашим конкретным требованиям. В последующих разделах вы узнаете, как можно настраивать политики.

Выбор столбцов для аудита

Если записывать информацию каждый раз, когда кто-то что-то выберет из таблицы, журнал аудита сильно разрастется, и им будет сложно управлять. Возможно, имеет смысл регистрировать доступ только к определенному набору столбцов. Давайте вернемся к описанию таблицы EMP.

```
SQL> DESC emp
 Name              Null?    Type
 ----------------- -------- ------------
 EMPID             NOT NULL NUMBER(4)
 EMPNAME                    VARCHAR2(10)
 JOB                        VARCHAR2(9)
 MGR                        NUMBER(4)
 HIREDATE                   DATE
 SALARY                     NUMBER(7,2)
 COMM                       NUMBER(7,2)
 DEPTNO                     NUMBER(2)
```

Если внимательно посмотреть на столбцы, то окажется, что для некоторых из них аудит можно считать более важным, чем для остальных. Например, все обращения к столбцу SALARY регистрировать необходимо, а значения столбца HIREDATE, возможно, не следует так же строго контролировать. Давайте предположим, что на этот раз необходим аудит доступа только к столбцам SALARY и COMM, но не ко всем остальным. Для этого следует задать значение параметра audit_column процедуры ADD_POLICY следующим образом:

```
BEGIN
   DBMS_FGA.add_policy (object_schema      => 'HR',
                        object_name        => 'EMP',
                        policy_name        => 'EMP_SEL',
                        audit_column       => 'SALARY, COMM'
                       );
END;
```

Такая настройка приводит к тому, что журнал аудита пишется только в случае, если пользователь выбирает данные из столбца SALARY или COMM. Если запрос обращается только к столбцу ENAME, журнал вестись не будет.

Все вышесказанное относится не только к столбцам, явно названным в запросе, но и к столбцам, на которые запрос ссылается неявно. Например, запрос

```
SELECT * FROM hr.emp;
```

выбирает все столбцы из таблицы EMP, включая COMM и SALARY. Поэтому это действие записывается. Несмотря на то что имена столбцов явно не названы, запрос ссылается на них неявно.

В Oracle9*i* детальный аудит запускается, как только встречается ссылка на *любой* из столбцов, перечисленных в параметре audit_column. В Oracle 10*g* существует возможность указать, следует ли включать аудит при ссылке на *один* из столбцов, или же только при ссылке на *все* столбцы (см. раздел «FGA для Oracle 10*g*»).

Выбор условий аудита

Предположим, что ваша компания является глобальной корпорацией с 50000 (или более) сотрудников, разбросанных по всему миру. Принимая во внимание различия трудовых законодательств и циклов оплаты, можно сказать, что база данных персонала HR работает преимущественно в режиме OLTP. Если в этом случае регистрировать любое обращение к столбцам COMM и SALARY, то журнал аудита очень скоро разрастется до неуправляемых размеров. Обдумывая возможные решения, вы можете захотеть ограничить регистрацию обращений только в выдающихся случаях (например, при попытке просмотра зарплат, превышающих 150000, или при попытке просмотра *вашей* личной зарплаты). Подобное ограничение можно задать в политике FGA, указав *условие* в специальном параметре audit_condition при вызове процедуры. Если политика уже была определена, удаляем ее:

```
BEGIN
   DBMS_FGA.drop_policy (object_schema      => 'ARUP',
                         object_name        => 'EMP',
                         policy_name        => 'EMP_SEL'
                        );
END;
```

Затем создаем новую политику:

```
BEGIN
   DBMS_FGA.add_policy (object_schema      => 'HR',
                        object_name        => 'EMP',
                        policy_name        => 'EMP_SEL',
                        audit_column       => 'SALARY, COMM',
                       audit_condition    => 'SALARY >= 150000 OR EMPID = 100'
                        );
END;
```

Параметр audit_condition использован для ограничения регистраций в журнале аудита только теми случаями, когда значение столбца SALARY превышает 150000 или когда значение EMPID равно 100. Если пользователь выбирает запись для кого-то, чья зарплата равна, например, 149999, такое действие не будет регистрироваться. Обратите внимание,

что для формирования записи журнала аудита необходимо выполнение *обоих* условий: пользователь должен обращаться к определенным столбцам *и* условие аудита должно быть выполнено. Если пользователь не обращается к значению столбца SALARY или COMM в запросе, то журнал не будет формироваться, даже если запрашиваемая запись имеет значение 150000 в столбце SALARY. Например, пусть зарплата Jake составляет 160000 и его идентификатор EMPID равен 52. Пользователь, который просто хочет узнать, кто является его начальником, выдает такой запрос:

```
SELECT mgr
FROM emp
WHERE empid = 52;
```

Пользователь не выбирает данные ни из столбца SALARY, ни из COMM, поэтому журнал не ведется. Однако запрос

```
SELECT mgr
FROM emp
WHERE salary > 160000;
```

формирует журнал. Столбец SALARY упомянут в предложении WHERE, поэтому пользователь неявно выбирает его, так что условие обращения к столбцу выполнено. Значение SALARY извлеченных записей превышает 150000 – условие аудита выполнено. Оба события произошли, поэтому будет сформирована запись в журнале аудита.

Для генерирования записи в журнале аудита необходимо наступление *двух* событий: условие аудита должно оцениваться как «истина» *и* пользователь должен выбирать соответствующие столбцы. При наступлении только одного события запись аудита не будет сформирована.

Условие аудита не обязательно должно ссылаться на столбцы таблицы, для которой определена политика; оно может ссылаться на и другие значения, в том числе на псевдостолбцы. Последняя возможность удобна тогда, когда вы хотите отслеживать действия не всех пользователей, а только некоторых. Предположим, что требуется регистрировать обращения к таблице EMP пользователя Scott. Определяем политику следующим образом:

```
BEGIN
   DBMS_FGA.add_policy (object_schema        => 'HR',
                        object_name          => 'EMP',
                        policy_name          => 'EMP_SEL',
                        audit_column         => 'SALARY, COMM',
                        audit_condition      => 'USER=''SCOTT'''
                       );
END;
```

Регистрироваться будут только действия пользователя Scott. Условие можно без труда заменить на более подходящее, например, указав USER IN (''SCOTT'', ''FRED''), чтобы включить аудит для Scott и Fred.

Возможно, требуется регистрировать все действия, которые выполняются по завершении рабочего дня: определяем политику следующим образом:

```
BEGIN
   DBMS_FGA.add_policy
      (object_schema        => 'HR',
       object_name          => 'EMP',
       policy_name          => 'EMP_AH',
       audit_column         => 'SALARY, COMM',
       audit_condition      =>
         'to_number(to_char(sysdate,''hh24'')) not between 09 and 17'
      );
END;
```

Регистрироваться будут все обращения к столбцам SALARY и COMM всех пользователей, выполненные вне временного интервала, начинающегося в 9 часов утра и заканчивающегося в 5 часов вечера. Все политики были названы по-разному и определены для одной таблицы – EMP, поэтому впоследствии в журнале аудита можно будет идентифицировать записи, вызванные применением каждой из политик. Например, чтобы узнать, какой пользователь обращался к записям EMP в нерабочее время, выполним такой запрос:

```
SELECT db_user, ....
FROM dba_fga_audit_trail
WHERE policy_name = 'EMP_AH';
```

Вы можете задать любое количество условий, удовлетворяющих конкретным требованиям аудита для вашей собственной базы данных.

Запись переменных связывания

Я подготовил отличную ловушку для Scott – пользователя, который любит просматривать зарплаты разных высокооплачиваемых руководителей нашей компании и мою в том числе). Теперь каждый раз, когда Scott захочет узнать размер такой зарплаты, этот факт будет записан в журнал аудита.

Однако предположим, что после всех этих тщательных приготовлений Scott почувствует, что «запахло жареным», и каким-то образом раскроет мой план. Будучи исключительно сообразительным, он изменяет свой запрос, используя переменную связывания и надеясь тем самым избежать аудита:

```
SQL> variable EMPID number
SQL> execute :EMPID := 100
SQL> SELECT salary FROM emp WHERE empid = :EMPID;
```

Но его попытка терпит неудачу, так как FGA собирает значения переменных связывания (в дополнение к тексту SQL-оператора). Эти значения можно увидеть в столбце SQL_BIND представления DBA_FGA_AUDIT_TRAIL. В предыдущем примере были бы записаны следующие данные:

```
SQL> SELECT sql_text,sql_bind FROM dba_fga_audit_trail;

SQL_TEXT                                        SQL_BIND
----------------------------------------------- ----------------------
select * from hr.emp where empid = :empid       #1(3):100
```

Обратите внимание, что переменные связывания записываются в формате

```
#1(3):100
```

где

`#1`

Указывает на то, что речь идет о первой переменной связывания. Если в запросе несколько переменных связывания, то последующие будут отображаться как `#2`, `#3` и т. д.

`(3)`

Указывает фактическую длину значения переменной связывания. В нашем примере Scott использовал значение `100`, поэтому длина равна `3`.

`:100`

Указывает фактическое значение переменной связывания. В данном случае это **100**.

Столбец `SQL_BIND` содержит строку значений, если использовано несколько переменных связывания. Например, если бы запрос был таким:

```
SQL> VARIABLE empid number
SQL> VARIABLE sal number
SQL> BEGIN
  2>    :empid := 100;
  3>    :sal := 150000;
  4> END;
  5> /

PL/SQL procedure successfully completed.

SQL> SELECT * from hr.emp WHERE empid = :empid OR salary > :sal;
```

то столбец `SQL_BIND` выглядел бы следующим образом:

```
#1(3):100 #2(5):15000
```

Переменные связывания идентифицируются позицией, значением и длиной значения.

Запись значений переменных связывания чрезвычайно важна, причем не только в целях контролируемости, но и для анализа шаблона доступа к данным, который сложно исследовать каким-то иным способом. Предположим, вы хотите определить наилучшую схему индексирования для хранилища данных. Следует на некоторое время включить детальный аудит для объектов, затем проанализировать значения столбцов `SQL_TEXT` и `SQL_BIND` журнала аудита и получить представление о ти-

пах значений, выбираемых пользователями. Такую информацию можно было бы также получить из представления V$SQL, но данное представление осуществляет выборку из разделяемого пула, из которого с течением времени старые операторы могут быть удалены. Журналы FGA сохраняются в таблице FGA_LOG$ до тех пор, пока администратор базы данных явно не удалит их. Поэтому журналы FGA обеспечивают более надежный механизм сбора текстов запросов и значений переменных связывания. Полученные сведения оказываются весьма полезны при разработке стратегии индексирования и секционирования.

Отключение записи переменных связывания

В некоторых случаях нет необходимости в записи в журнал аудита SQL-текста и значений переменных связывания. Тогда можно отключить регистрацию таких значений для экономии пространства. Для этого (т. е., чтобы не записывать значения переменных связывания) следует установить параметр audit_trail процедуры ADD_POLICY пакета DBMS_FGA в значение DB (прекращение сбора текстов SQL и значений переменных связывания) вместо DB_EXTENDED (сбор текстов SQL и значений переменных связывания). Значение по умолчанию – DB_EXTENDED. Напишем PL/SQL-блок для отключения сбора текстов SQL и значений переменных связывания (параметру audit_trail присваиваем значение константы DBMS_FGA.DB):

```
BEGIN
   DBMS_FGA.add_policy (object_schema    => 'HR',
                        object_name      => 'EMP',
                        policy_name      => 'EMP_SEL',
                        audit_column     => 'SALARY, COMM',
                        audit_condition  => 'SALARY >= 150000 OR EMPID = 100',
                        audit_trail      => DBMS_FGA.db
                       );
END;
/
```

Задание модуля обработки

Вы уже видели, как FGA может регистрировать факт выборки (SELECT) данных (в Oracle9*i*) из таблицы или факт выполнения оператора DML (в Oracle 10*g*) в таблице журнала аудита FGA, FGA_LOG$. FGA поддерживает еще одну важную функцию – возможность выполнения хранимой PL/SQL-программы (хранимой процедуры на PL/SQL или Java-метода). Если хранимая процедура, в свою очередь, инкапсулирует программу оболочки (shell) или операционной системы, то и она может быть выполнена. Такая единица хранимой программы называется *модулем обработки* (*handler module*). В предыдущем примере при создании механизма аудита для обращений к таблице EMP можно дополнительно указать хранимую процедуру для выполнения (отдельную или в составе пакета). Например, для выполнения хранимой про-

цедуры myproc, принадлежащей пользователю FGA_ADMIN, просто вызываем процедуру ADD_POLICY с двумя новыми параметрами:

```
BEGIN
   DBMS_FGA.add_policy (object_schema    => 'HR',
                        object_name      => 'EMP',
                        policy_name      => 'EMP_SEL',
                        audit_column     => 'SALARY, COMM',
                       audit_condition  => 'SALARY >= 150000 OR EMPID = 100',
                        handler_schema   => 'FGA_ADMIN',
                        handler_module   => 'MYPROC'
                       );
END;
```

Если условия политики аудита выполнены, и встречается ссылка на указанные столбцы, то происходят два события: действие регистрируется в журнале аудита, и выполняется процедура myproc в схеме FGA_ADMIN. Процедура автоматически выполняется в рамках автономной транзакции при каждой вставке записи в журнал аудита. Это означает, что любые изменения, внесенные модулем обработки, будут зафиксированы или отменены без оказания какого-либо воздействия на транзакцию сеанса, вызвавшего модуль обработки. Они также не будут мешать выполнению аудита.

Если по какой-то причине модуль обработки не смог отработать корректно, FGA не сообщает об ошибке при выборке данных из таблицы. Вместо этого FGA просто молча прекращает извлечение строк, для которых не удалось выполнить модуль обработки. Это достаточно коварная ситуация, так как вы никак не узнаете о том, что модуль обработки не отработал. Будут возвращены не все строки, то есть результат будет неверным. Поэтому чрезвычайно важно тщательно тестировать такие модули!

Модуль обработки особенно полезен, если вы хотите записывать данные в собственные таблицы, а не только в обычные таблицы журнала аудита. На первый взгляд может показаться, что речь идет о простом повторении уже имеющейся функциональности. Однако в следующих разделах будет показано, что тем самым вы можете значительно расширить свои возможности.

Недостатки настроек FGA по умолчанию

Если вы помните, при детальном аудите журналы аудита пишутся в таблицу FGA_LOG$ схемы SYS табличного пространства SYSTEM. Такой механизм имеет три потенциально слабых места.

1. Так как эта таблица содержит сведения обо всех обращениях ко всем таблицам, для которых определен детальный аудит, ее содержимое является конфиденциальным и должно быть защищено. Однако пользователь SYS или любой другой пользователь с ролью администратора базы данных может без труда удалять строки из этой

таблицы, удаляя тем самым записи журнала FGA. Для чрезвычайно требовательных к безопасности организаций (в особенности тех, чья деятельность регулируется специальными нормативами, такими как HIPAA, Sarbanes-Oxley, Visa Cardholder Information Security Policy) возможность подделки данных аудита недопустима. Обеспечение безопасности для журнала аудита является первостепенной задачей, а механизм аудита «по умолчанию» этого не гарантирует.

2. Журналы аудита ведутся не только для операторов DML, но и для SELECT, поэтому за короткое время может быть сгенерировано большое количество записей. Частота обращений к базе данных и количество заданных параметров аудита определяют, насколько серьезными могут быть вызванные этим проблемы. В базе данных OLTP журналы аудита (расположенные в схеме SYS и табличном пространстве SYSTEM) могут существенно «раздуть» табличное пространство SYSTEM. Даже если таблица будет усекаться после периодического архивирования, может оказаться, что увеличившиеся файлы данных не могут сократиться до прежнего размера, и в конце концов в табличном пространстве SYSTEM будет масса неиспользуемого места.

   Альтернативный подход заключается в создании пользовательского журнала аудита и помещении его в пользовательское табличное пространство, где им так же удобно управлять, как и любым другим табличным пространством. Можно также специально спроектировать таблицу таким образом, чтобы оптимизировать производительность и архивацию. Например, можно секционировать таблицу, сделав управление более гибким.

3. При настройке по умолчанию записи просто вносятся в журнал, никакого уведомления о возникшей проблеме никому не отсылается. Oracle не поддерживает возможность создания триггеров для таблицы журнала аудита, которые можно было бы использовать для отправки электронных сообщений или предупреждений. Однако возможность автоматического вызова хранимой программы может быть весьма полезной, если необходимо выполнить какое-то предопределенное действие при выполнении условий аудита. Например, когда кто-то рассматривает данные о зарплате наиболее высокооплачиваемых топ-менеджеров вашей компании, можно отправлять уведомление сотруднику службы безопасности посредством Oracle Advanced Queuing или Oracle Streams. Можно также отправить электронное сообщение при наступлении определенного события или зарегистрировать факты доступа к данным сотрудников высокого ранга в специальной таблице.

Пользовательская настройка аудита

Решить возможные проблемы может пользовательская настройка аудита. Создадим таблицу для хранения записей. Она могла бы входить в любую схему, но из соображений безопасности поместим ее в схему,

которая для других целей использоваться не будет. Будем использовать ту же схему, что и раньше: FGA_ADMIN. Вот пример такой таблицы:

```
/* Файл на веб-сайте: cr_flagged_access.sql */

CREATE TABLE flagged_access
(
   fgasid            NUMBER(20),
   entryid           NUMBER(20),
   audit_date        DATE,
   fga_policy        VARCHAR2(30),
   db_user           VARCHAR(30),
   os_user           VARCHAR2(30),
   authent_type      VARCHAR2(30),
   client_id         VARCHAR2(100),
   client_info       VARCHAR2(64),
   host_name         VARCHAR2(54),
   instance_id       NUMBER(2),
   ip                VARCHAR2(30),
   term              VARCHAR2(30),
   schema_owner      VARCHAR2(20),
   table_name        VARCHAR2(30),
   sql_text          VARCHAR2(64),
   SCN               NUMBER(10)
)
TABLESPACE audit_ts
PARTITION BY RANGE (audit_date)
(
   PARTITION y04m01 VALUES LESS THAN
      (TO_DATE('02/01/2004','mm/dd/yyyy')),
   PARTITION y04m02 VALUES LESS THAN
      (TO_DATE('03/01/2004','mm/dd/yyyy')),
   PARTITION y04m03 VALUES LESS THAN
      (TO_DATE('04/01/2004','mm/dd/yyyy')),
   PARTITION y04m04 VALUES LESS THAN
      (TO_DATE('05/01/2004','mm/dd/yyyy')),
   PARTITION y04m05 VALUES LESS THAN
      (TO_DATE('06/01/2004','mm/dd/yyyy')),
   PARTITION y04m06 VALUES LESS THAN
      (TO_DATE('07/01/2004','mm/dd/yyyy')),
   PARTITION y04m07 VALUES LESS THAN
      (TO_DATE('08/01/2004','mm/dd/yyyy')),
   PARTITION y04m08 VALUES LESS THAN
      (TO_DATE('09/01/2004','mm/dd/yyyy')),
   PARTITION def VALUES LESS THAN
      (MAXVALUE)
);
```

Стратегия очистки данной таблицы чрезвычайно проста: каждый месяц все данные аудита, срок хранения которых превышает один месяц, перемещаются в отдельное автономное хранилище и удаляются из таблицы. Таким образом, происходит помесячное диапазонное секционирование таблицы по столбцу AUDIT_DATE. Когда приходит время очист-

ки, я преобразую секцию в таблицу, используя команду ALTER TABLE…EXCHANGE PARTITION, а затем пересылаю содержимое на магнитную ленту при помощи такой функциональности Oracle, как Transportable Tablespace (переносимые табличные пространства). Затем удаляю раздел. Каждый месяц создаются новые секции для наступающих месяцев.

Теперь необходимо создать процедуру для заполнения таблицы. Будем работать в той же безопасной схеме, где хранится таблица (например, FGA_ADMIN). Процедура будет вызывать встроенную функцию DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER, поэтому необходимо явно выдать привилегию EXECUTE на соответствующий пакет. От имени пользователя SYS выполняем:

```
GRANT EXECUTE ON dbms_flashback TO fga_admin;
```

Теперь создаем процедуру.

```
/* Файл на веб-сайте: access_flagger.sql *

 1  CREATE OR REPLACE PROCEDURE access_flagger
 2  (
 3     p_table_owner    IN VARCHAR2,
 4     p_table_name     IN VARCHAR2,
 5     p_fga_policy     IN VARCHAR2
 6  )
 7  IS
 8     l_fgasid       NUMBER(20);
 9     l_entryid      NUMBER(20);
10     l_term         VARCHAR2(2000);
11     l_db_user      VARCHAR2(30);
12     l_os_user      VARCHAR2(30);
13     l_authent_type VARCHAR2(2000);
14     l_client_id    VARCHAR2(100);
15     l_client_info  VARCHAR2(64);
16     l_host_name    VARCHAR2(30);
17     l_instance_id  NUMBER(2);
18     l_ip           VARCHAR2(30);
19     l_sql_text     VARCHAR2(4000);
20     l_scn          NUMBER;
21  BEGIN
22     l_fgasid       := sys_context('USERENV','SESSIONID');
23     l_entryid      := sys_context('USERENV','ENTRYID');
24     l_term         := sys_context('USERENV','TERMINAL');
25     l_db_user      := sys_context('USERENV','SESSION_USER');
26     l_os_user      := sys_context('USERENV','OS_USER');
27     l_authent_type := sys_context('USERENV','AUTHENTICATION_TYPE');
28     l_client_id    := sys_context('USERENV','CLIENT_IDENTIFIER');
29     l_client_info  := sys_context('USERENV','CLIENT_INFO');
30     l_host_name    := sys_context('USERENV','HOST');
31     l_instance_id  := sys_context('USERENV','INSTANCE');
32     l_ip           := sys_context('USERENV','IP_ADDRESS');
33     l_sql_text     := sys_context('USERENV','CURRENT_SQL');
34     l_scn          := SYS.DBMS_FLASHBACK.get_system_change_number;
```

```
35     INSERT INTOS flagged_access
36     (
37        fgasid,
38        entryid,
39        audit_date,
40        fga_policy,
41        db_user,
42        os_user,
43        authent_type,
44        client_id,
45        client_info,
46        host_name,
47        instance_id,
48        ip,
49        term,
50        schema_owner,
51        table_name,
52        sql_text,
53        scn
54     )
55     VALUES
56     (
57        l_fgasid,
58        l_entryid,
59        sysdate,
60        p_fga_policy,
61        l_db_user,
62        l_os_user,
63        l_authent_type,
64        l_client_id,
65        l_client_info,
66        l_host_name,
67        l_instance_id,
68        l_ip,
69        l_term,
70        p_table_owner,
71        p_table_name,
72        l_sql_text,
73        l_scn
74     );
75* END;
76 /
```

Ключевые моменты кода поясняются в таблице.

Строки	Описание
3–5	Обратите внимание на входные параметры. В модуле обработки необходимо использовать именно приведенные здесь параметры, а не какие-то другие, и именно в указанном порядке: владелец таблицы, имя таблицы и имя политики FGA. Конечно, имена параметров могут быть любыми, но их позиция определяет их назначение.

Строки	Описание
22–34	Эти строки содержат разнообразную информацию о действиях и сеансах пользователя. Их может быть больше, – они могут использовать всю информацию, предоставляемую контекстом USERENV.
34	Возвращенный функцией текущий системный номер изменения SCN записывается в журнал аудита.
35	Собранные значения вставляются в созданную ранее таблицу FLAGGED_ACCESS.

Из-за ошибки Oracle вызов функции SYS_CONTEXT('USERENV','SESSIONID') всегда возвращает 0 внутри обработчика FGA. Эта ошибка была исправлена в версии Oracle 10*g* Release 2. На момент создания книги не существовало патча, исправляющего эту ошибку для предшествующих версий, поэтому строка 2 всегда будет иметь значение 0 (пока не выйдет и не будет установлен соответствующий патч).

Наконец, добавляем процедуру в политику FGA, чтобы она вызывалась автоматически при выполнении условий аудита:

```
BEGIN
   DBMS_FGA.add_policy (object_schema    => 'HR',
                        object_name      => 'EMP',
                        policy_name      => 'EMP_SEL',
                        audit_column     => 'SALARY, COMM',
                        audit_condition  => 'SALARY >= 150000 OR EMPID = 100',
                        handler_schema   => 'FGA_ADMIN',
                        handler_module   => 'ACCESS_FLAGGER'
                       );
END;
/
```

Теперь при выполнении условий аудита запись вносится в таблицу FGA_LOG$, а также в таблицу FLAGGED_ACCESS. Другими словами, пользовательский обработчик аудита не отменяет ведение *обычного* журнала аудита. Причина проста: модуль обработки служит не только для создания пользовательского журнала аудита, он также используется для автоматического вызова хранимой процедуры, которая может выполнять разнообразные действия: отправлять электронные сообщения, выдавать предупреждения, обновлять какой-то флаг и т. д. Поэтому системные журналы аудита необходимо поддерживать в любом случае.

Если необходимо обеспечить еще *более* надежную защиту хранимой информации, можно создать идентичную таблицу в удаленной базе данных и использовать одностороннее создание моментальных копий в режиме «только для чтения». Для любой записи, вставленной в таблицу FLAGGED_ACCESS локальной базы данных, создается идентичная строка в удаленной таблице. Но и здесь возможны злоупотребления со стороны администратора базы данных, который может удалить журнал

моментальной копии. Однако в базах данных с относительно невысокой активностью можно задать короткий период обновления, чтобы значения передавались сразу после их появления в исходной базе данных и у администратора было очень мало времени для удаления записей. К сожалению, такое решение является единственно доступным, так как внутри модуля обработки распределенные транзакции не поддерживаются, поэтому невозможно выполнить вставку напрямую в удаленную таблицу.

Даже если записи удалены, можно получить значения из архивных файлов журнала при помощи Oracle Log Miner. Так защищают журналы FGA в высоконадежных системах.

Можно использовать модуль обработки не только для создания пользовательского журнала аудита, но и для выполнения других операций (например, для отправки электронного сообщения ревизору при обращении к особо секретным данным вне рабочего времени).

Администрирование FGA

Пока мы говорили только о том, как задать детальный аудит для отдельной таблицы. Администратор базы данных отвечает также за контроль и управление всей конфигурацией FGA.

Представление DBA_AUDIT_POLICIES

Для просмотра политик FGA, которые уже были определены в базе данных, можно обратиться к представлению словаря базы данных `DBA_AUDIT_POLICIES`, которое содержит следующие столбцы:

`OBJECT_SCHEMA`

Имя схемы, которой принадлежит таблица или представление, для которых определена политика.

`OBJECT_NAME`

Имя таблицы или представления, для которых определена политика.

`POLICY_NAME`

Имя политики.

`POLICY_TEXT`

Если существует условие политики, например `SALARY>=150000 OR EMPID=100`, оно приводится в данном столбце.

`POLICY_COLUMN`

Если политика связана с определенными столбцами (например, журнал аудита формируется только при ссылке на определенные столбцы), их названия выводятся в данном столбце.

PF_SCHEMA

Если определен модуль обработки для политики, отображается имя его владельца.

PF_FUNCTION

Если модуль обработки определен и является независимой процедурой, выводится его имя. Если модуль обработки является пакетной процедурой, то в данном столбце выводится имя процедуры, а в столбце PF_PACKAGE – имя пакета.

PF_PACKAGE

См. описание столбца PF_FUNCTION.

ENABLED

Указывает, включена ли политика аудита (YES или NO).

В Oracle 10*g* данное представление содержит ряд дополнительных столбцов:

SEL

INS

UPD

DEL

AUDIT_TRAIL

POLICY_COLUMN_OPTIONS

Они будут описаны далее в разделе «FGA в Oracle 10*g*».

Использование процедур DBMS_FGA

DBMS_FGA – достаточно простой пакет, включающий в себя всего 4 процедуры, которые можно использовать для администрирования детального аудита базы данных: ADD_POLICY, DROP_POLICY, DISABLE_POLICY и ENABLE_POLICY.

Процедура ADD_POLICY

Помните, что политика FGA не является *объектом* базы данных, поэтому к ней не применяются обычные операции SQL. Для администрирования политики необходимо использовать процедуру ADD_POLICY пакета DBMS_FGA, которая уже была описана выше.

Процедура DROP_POLICY

Для удаления политики FGA следует использовать процедуру DROP_POLICY пакета DBMS_FGA. Например, для удаления определенной ранее политики для таблицы EMP необходим такой фрагмент кода:

```
BEGIN
   DBMS_FGA.drop_policy (object_schema      => 'HR',
                         object_name        => 'EMP',
```

```
                          policy_name        => 'EMP_SEL'
                         );
END;
/
```

Если политика не существует, будет сгенерирована ошибка: «ORA-28102: policy does not exist».

Процедура DISABLE_POLICY

Может возникнуть необходимость временного отключения записи журнала аудита, например для очистки журнала аудита или изменения таблицы для его хранения. Удалять политику не придется, можно просто отключить ее, используя процедуру DISABLE_POLICY пакета DBMS_FGA:

```
BEGIN
   DBMS_FGA.disable_policy (object_schema      => 'HR',
                            object_name        => 'EMP',
                            policy_name        => 'EMP_SEL'
                           );
END;
/
```

Процедура ENABLE_POLICY

Отключенная политика сохраняется, только не ведется запись в журнал аудита. После того как необходимые манипуляции по обслуживанию выполнены, можно снова включить политику, используя процедуру ENABLE_POLICY пакета DBMS_FGA:

```
BEGIN
   DBMS_FGA.enable_policy (object_schema      => 'HR',
                           object_name        => 'EMP',
                           policy_name        => 'EMP_SEL'
                          );
END;
/
```

FGA в Oracle 10*g*

В этом разделе описаны специальные возможности и расширения функциональности FGA в версии Oracle 10*g*.

Дополнительные команды DML

В Oracle9*i* FGA поддерживает аудит только для оператора SELECT; такие операторы DML, как INSERT, UPDATE и DELETE, детальному аудиту подвергнуть невозможно. В Oracle 10*g* детальный аудит доступен и для операторов DML. Новый параметр statement_types процедуры ADD_POLICY пакета DBMS_FGA позволяет указать, для каких операторов следует проводить аудит. Продолжая предыдущий пример, предположим, что теперь мы хотим собирать сведения обо всех типах операторов: SELECT, IN-

SERT, UPDATE и DELETE, для таблицы EMP, но только в том случае, если выполнены условия аудита. Этого можно достичь следующим образом:

```
BEGIN
   DBMS_FGA.add_policy (object_schema        => 'HR',
                        object_name          => 'EMP',
                        policy_name          => 'EMP_DML',
                        audit_column         => 'SALARY, COMM',
                        audit_condition      => 'SALARY >= 150000',
                        statement_types      => 'SELECT, INSERT, DELETE,
                                                 UPDATE'
                       );
END;
/
```

Записи попадают все в ту же таблицу FGA_LOG$ и доступны через то же представление словаря данных – DBA_FGA_AUDIT_TRAIL. Чтобы учесть три дополнительных типа обращения к данным (INSERT, UPDATE и DELETE), в представление добавлен новый столбец STATEMENT_TYPE. Если новый параметр процедуры опущен, то регистрируются только операторы SELECT.

Для операторов DELETE аудит ведется всегда, вне зависимости от параметра audit_column. Дело в том, что DELETE удаляет всю строку и неявно ссылается (воздействует) на все столбцы таблицы.

Только простые предикаты (то есть содержащие всего одно условие) могут использоваться в параметре audit_condition для операторов DML. Нельзя, например, определить политику для детального аудита операторов DML, задав параметр audit_condition следующим образом: 'SALARY >= 150000 OR EMPID = 100'. Политику создать удастся, но обновления таблицы будут завершаться с ошибкой «ORA-28138: Error in Policy Predicate» (ошибка в предикате политики). Аналогично в audit_condition нельзя задать подзапрос. Разрешены только простые предикаты.

Дополнительные представления словаря данных и столбцы

В Oracle 10*g* в журнал детального аудита записывается ряд дополнительных элементов, которые доступны через представление DBA_FGA_AUDIT_TRAIL. Кроме того, появляется дополнительное представление FLASHBACK_TRANSACTION_QUERY.

Представление DBA_FGA_AUDIT_TRAIL

В Oracle 10*g* в представление DBA_FGA_AUDIT_TRAIL добавляются следующие новые столбцы:

STATEMENT_TYPE

Тип оператора, для которого выполнена регистрация (например, SELECT, INSERT, UPDATE или DELETE).

EXTENDED_TIMESTAMP

В дополнение к обычной временной метке хранится расширенная, включающая в себя микросекунды, часовой пояс и другую дополнительную информацию.

PROXY_SESSIONID

Для прокси-пользователя, получившего привилегию на доверенный вход в систему, как показано ниже, идентификатор прокси-сеанса сохраняется в данном столбце:

```
ALTER USER seeta GRANT CONNECT THROUGH geeta;
```

GLOBAL_UID

Если пользователь является корпоративным пользователем, например, определенным через LDAP, то идентификатор пользователя отличается от обычного идентификатора пользователя Oracle. В этот столбец записывается идентификатор корпоративного или глобального пользователя.

INSTANCE_NUMBER

При использовании технологии Real Application Clusters (RAC) каждому экземпляру может соответствовать свой идентификатор сеанса, то есть пользовательский сеанс однозначно идентифицируется комбинацией номера экземпляра и идентификатора сеанса. В Oracle 10*g*, в отличие от предыдущих версий, номер экземпляра регистрируется. (Обратите внимание, что в приведенном выше пользовательском журнале аудита мы собирали эти сведения.)

OS_PROCESS

Идентификатор процесса операционной системы для пользовательского сеанса. Этот идентификатор оказывается чрезвычайно важен, если необходим анализ соответствующего файла трассировки.

TRANSACTIONID

Идентификатор транзакции, используемый в ретроспективных запросах (для того чтобы понять, что за значение хранится в данном столбце, необходимо представлять себе ретроспективные запросы, о которых мы говорили ранее в главе).

Представление FLASHBACK_TRANSACTION_QUERY

В Oracle 10*g* появляется новое представление словаря данных, FLASHBACK_TRANSACTION_QUERY, которое отображает выполненные в базе данных транзакции. Представление содержит следующие столбцы:

XID

Каждая транзакция имеет уникальный номер, который записывается в данный столбец в виде значения типа RAW.

START_SCN

Системный номер изменения (SCN) на момент начала транзакции.

START_TIMESTAMP

Временная метка на момент начала транзакции.

COMMIT_SCN

Номер SCN на момент фиксации транзакции.

COMMIT_TIMESTAMP

Временная метка на момент фиксации транзакции.

LOGON_USER

Пользователь Oracle, который открыл сеанс.

UNDO_CHANGE#

Номер SCN операции отката.

OPERATION

Тип операции (например, INSERT, UPDATE, DELETE). Если речь идет о PL/SQL-блоке, то в столбце выводится DECLARE или BEGIN.

TABLE_NAME

Имя таблицы, упоминаемой в транзакции.

TABLE_OWNER

Владелец вышеупомянутой таблицы.

ROW_ID

Идентификатор строки, которая была изменена или прочитана.

UNDO_SQL

Оператор SQL, который может отменить изменения, выполненные исходным оператором. Если исходный оператор SQL – INSERT, то в столбце UNDO_SQL хранится DELETE, и т. д.

Информация данного представления полезна в основном в рамках использования функциональности ретроспективных запросов (новой возможности Oracle 10*g*), но может пригодиться и для детального аудита. Столбец XID данного представления содержит уникальный идентификатор транзакции, который также хранится в столбце TRANSACTIONID представления DBA_FGA_AUDIT_TRAIL. На основе этого столбца можно установить соответствие между двумя представлениями и получить всю информацию о транзакции, которая привела к регистрации записи в журнале аудита.

Комбинация столбцов

В предыдущем примере список столбцов задавался следующим образом:

```
audit_column    => 'SALARY, COMM'
```

Это означает, что будет вестись аудит при обращении пользователя к столбцу SALARY или COMM. Однако в некоторых случаях возникает тре-

бование вести аудит только при ссылке на *все* столбцы списка, а не на какой-то *один* из них. Например, в базе данных EMP можно вести журнал только в том случае, если кто-то *одновременно* запрашивает данные из столбцов SALARY и EMPNAME. Дело в том, что при доступе только к одному столбцу раскрытие конфиденциальной информации маловероятно (обычно пользователь ищет зарплату по имени сотрудника). Предположим, что пользователь выдает такой запрос:

```
SELECT salary FROM hr.emp;
```

Будут выведены зарплаты всех сотрудников, но без указания того, кому какая цифра соответствует. Такая информация, скорее всего, не представляет особого интереса. Предположим теперь, что выдан такой запрос:

```
SELECT empname FROM hr.emp;
```

Выдаются имена сотрудников, но без указания их зарплат, сведения о зарплате защищены. Но если пользователь выполняет следующий запрос:

```
SELECT empname, salary FROM hr.emp;
```

то *будет* выведена информация о зарплатах с указанием сотрудников. Это как раз те данные, которые хотелось бы защитить. И в этом последнем случае (но не в двух первых) журнал аудита содержит полезную информацию, поэтому его стоит вести.

В версии Oracle9*i* не существовало возможности указать в условии аудита комбинацию столбцов. В Oracle 10*g* это можно сделать при помощи параметра audit_column_opts процедуры ADD_POLICY. По умолчанию этот параметр установлен в значение DBMS_FGA.ANY_COLUMNS, что означает включение аудита при обращении к любому из столбцов списка. Если заменить значение по умолчанию на DBMS_FGA.ALL_COLUMNS, то журнал аудита будет формироваться только при обращении ко всем столбцам из списка. В нашем случае создаем политику FGA, которая будет создавать запись аудита только в том случае, когда пользователь выбирает данные из обоих столбцов: SALARY и EMPNAME, следующим образом:

```
BEGIN
   DBMS_FGA.add_policy (object_schema          => 'HR',
            object_name           => 'EMP',
            policy_name           => 'EMP_DML',
            audit_column          => 'SALARY, COMM',
            audit_condition       => 'SALARY >= 150000 OR EMPID = 100',
            statement_types       => 'SELECT, INSERT, DELETE, UPDATE',
            audit_column_opts     => DBMS_FGA.all_columns
             );
END;
/
```

Такая возможность позволяет сконцентрировать усилия на отслеживании только ключевых условий (и уменьшить объем журнала аудита).

Для поддержки новой функциональности в представление DBA_AUDIT_POLICIES в Oracle 10*g* добавлен ряд дополнительных столбцов.

SEL

Указывает, запускает ли политика ведение журнала аудита FGA в случае выборки (SELECT) данных (YES или NO).

INS

То же самое, но для INSERT.

UPD

То же самое, но для UPDATE.

DEL

То же самое, но для DELETE.

AUDIT_TRAIL

Столбцы SQL_TEXT и SQL_BIND заполняются только в том случае, если параметр AUDIT_TRAIL установлен в значение DB_EXTENDED (умолчание), а не в DB. Данный столбец отображает значение этого параметра.

POLICY_COLUMN_OPTIONS

Указывает, ведется ли журнал аудита при ссылке на все столбцы списка или на любой из них.

FGA и другие технологии аудита Oracle

В этом разделе мы сравним FGA с другими технологиями аудита Oracle, в частности с триггерами и обычным аудитом, и рассмотрим причины, по которым использование какого-то из способов может оказаться предпочтительным.

Детальный аудит и триггеры

Изменения в базе данных, вызванные выполнением операторов DML, традиционно отслеживались с помощью триггеров. Триггеры уровня строки для таких операторов DML, как INSERT, UPDATE и DELETE, могут регистрировать имя пользователя, временную метку, измененный объект и другие сведения. Приведем пример записи изменений в таблице EMP при помощи триггера:

```
CREATE OR REPLACE TRIGGER tr_ar_iud_emp
   AFTER INSERT OR DELETE OR UPDATE
   ON emp
   FOR EACH ROW
BEGIN
   INSERT INTO audit_trail
        VALUES (USER, SYSDATE, 'EMP', USERENV ('ip_address')
              -- ... и так далее ...
               );
END;
/
```

Начиная с версии Oracle 10*g* FGA также может регистрировать такие сведения для операторов DML: временную метку изменения, IP-адрес и многое другое. Исчезает ли в связи с этим необходимость в триггерах? Не совсем. В следующих разделах будет показано, что оба подхода имеют свои плюсы и минусы.

Детальный аудит

Сначала поговорим о детальном аудите. FGA имеет несколько явных преимуществ по сравнению с триггерами.

Отслеживание не-DML запросов

Первое преимущество уже неоднократно отмечалось в этой главе. FGA позволяет вести записи для пользователей, которые просто выбирают данные, не изменяя их. Триггеры не запускаются при выборке данных, поэтому они бесполезны для аудита операторов `SELECT`.

Простота программирования

Триггеры требуют написания большого объема кода, соответствующего вашим конкретным требованиям, который затем еще нужно поддерживать. Со своей стороны, FGA не требует больших усилий от программиста.

Регистрируемые события

Триггеры запускаются только при изменении данных, в то время как FGA ведет журнал вне зависимости от того, менялись ли данные. Если данные были сначала изменены, затем изменение было откачено, то триггеры не зарегистрируют изменения (если только вы не используете автономные транзакции). FGA вносит записи в журнал аудита посредством автономных транзакций, поэтому обращения к данным будут записаны. В системах с высокой защищенностью такое требование может быть обязательным.

Количество записей журнала

При выполнении изменения триггеры уровня строки вносят в журнал аудита по одной записи для каждой измененной строки. В случае с FGA для каждого оператора в журнал аудита записывается всего одна строка, вне зависимости от того, сколько строк затронуто, что уменьшает объем журнала.

Выбор столбцов

Если вы хотите отслеживать доступ только к некоторым столбцам, то можно использовать предложение `WHERE` триггера. Например, для записи обращений только к столбцу `SALARY`, изменим триггер следующим образом:

```
CREATE OR REPLACE TRIGGER tr_ar_iud_emp
   AFTER INSERT OR DELETE OR UPDATE
   ON emp
   FOR EACH ROW
```

```
   WHEN (:NEW.salary != :OLD.salary)
BEGIN
   INSERT INTO audit_trail
        VALUES (USER, SYSDATE, 'EMP', USERENV ('ip_address'),
                  -- ... и так далее ...
                );
END;
/
```

Для FGA просто задаем параметр:

```
audit_columns => 'SALARY'
```

Результат в обоих случаях будет одним и тем же: изменения будут регистрироваться только при обращении к столбцу SALARY. Но есть одно важное отличие. Если столбец SALARY не изменяется, а просто упоминается в предикате, то FGA будет учитывать этот факт, а триггеры – нет.

Представления

Теперь предположим, что существует представление VW_EMP для таблицы EMP. Это представление, в отличие от таблицы, доступно конечным пользователям, так что вы могли бы определить набор триггеров INSTEAD OF для манипулирования таблицей в случаях, когда пользователи будут манипулировать данными представления. Однако если триггер INSTEAD OF изменит элемент данных, то определенные для таблицы триггеры аудита не смогут увидеть это изменение и соответственно не зарегистрируют его. FGA регистрирует любые изменения вне зависимости от того, как они были совершены.

Триггеры

Всегда ли FGA отслеживает небольшие изменения лучше, чем триггеры? Это не совсем так. В большинстве случаев FGA с легкостью обеспечивает механизм детального аудита, необходимый для удовлетворения ваших требований. Однако в некоторых случаях триггеры работают лучше, чем FGA:

Отсутствие необходимости выделения большого сегмента отката

Когда пользователь Scott изменяет элемент данных, как можно узнать, каким было значение до изменения? Ранее уже говорилось о том, что можно использовать ретроспективные запросы Oracle для получения старого значения из таблицы с помощью номера SCN, записанного в журнале аудита. Например, для определения того, каким было значение SALARY до того, как Scott его изменил, следует выполнить такой запрос:

```
SELECT salary FROM emp AS OF scn 14122310350 WHERE empid = 100;
```

Изменение произошло 6 июня, а сегодня 20 июня, т. е. прошло 14 дней после изменения. Ретроспективный запрос использует для восстановления данных сегменты отката, так что данные должны

там присутствовать. Для того чтобы можно было вернуться к данным по состоянию на 14 дней назад, параметр инициализации UNDO_RETENTION_PERIOD должен быть установлен в 14 дней. Для этого в свою очередь необходимо, чтобы табличное пространство отката было достаточно большим, чтобы вмещать в себя такой объем данных. В большинстве компаний выделение такого большого табличного пространства может быть весьма проблематичным. Дополнительную сложность создает тот факт, что при остановке базы данных данные отката пропадают.

Если же вместо FGA использовать для отслеживания подобных изменений триггеры, то старые значения можно легко сохранить в самом журнале аудита, и огромное табличное пространство отката не понадобится. Кроме того, старые значения «выживут» и при остановке базы данных. Так что в подобных ситуациях, вероятно, разумнее применять триггеры, а не FGA.

Хранение выборочных данных

Данные отката собираются не только для интересующих вас изменений базы данных, но и для всех вообще. В активно используемых базах данных может формироваться значительный объем данных отката, и может случиться так, что необходимая информация будет вычищена из сегментов отката до того, как ее удастся использовать в ретроспективном запросе.

Как уже говорилось в предыдущем пункте, при работе с триггерами данные будут сразу сохранены в таблицах, и никакая очистка им не страшна.

Отсутствие записей аудита при откате

При определенных обстоятельствах FGA формирует гораздо более объемный и гораздо менее полезный журнал аудита, чем тот, что можно получить при использовании триггеров. Пусть пользователь Scott выполняет такой оператор DML:

```
UPDATE hr.emp SET salary = 14000 WHERE empid = 100;
```

Если для таблицы EMP определена политика FGA, для этого оператора будет сформирована запись в журнале аудита. Так как журнал FGA ведется в режиме автономных транзакций, запись в журнале аудита фиксируется вне зависимости от того, что произошло в транзакции пользователя Scott. Предположим, что Scott откатывает, а не фиксирует изменение. Фактически строка не была обновлена, однако запись в журнале зафиксирована и не откатывается. В результате в журнале аудита присутствуют ложные результаты.

Если использовать триггеры, то вставка записи в журнал аудита будет частью той же самой транзакции, так что она будет откачена при откате основной транзакции, и ошибочных записей не возникнет. Если система выполняет множество изменений и часто их откатывает, то журнал FGA станет невероятно большим, при этом

большая его часть будет представлять собой ложные данные. Разумнее использовать триггеры.

Очевидно, что FGA не может полностью заменить собой триггеры, каждый подход имеет свою нишу. Принимая решение об использовании FGA, внимательно проанализируйте приведенные отличия и посмотрите, какой из способов лучше подходит в каждой конкретной ситуации.

FGA и обычный аудит

Начиная с Oracle 10*g* Release 1 обычный аудит Oracle (реализуемый командой `AUDIT`) был расширен и обеспечивает регистрацию информации, которая была недоступна в предыдущих версиях (например, текст выданного оператора SQL, переменные связывания и т. д.). Он во многом стал похож на FGA. Избавляет ли он при этом от необходимости использования FGA? Не совсем. Давайте рассмотрим различия между обычным и детальным аудитом.

Типы операторов

Обычный аудит может отслеживать множество различных видов операторов: DML, DDL, операторы управления сеансом, операторы управления привилегиями и т. д. FGA может отслеживать только один оператор (`SELECT`) в версии Oracle9*i* и четыре (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) – в Oracle 10*g*.

Специальные параметры

FGA запускается, не требуя никаких специальных параметров. Таблица `FGA_LOG$` – хранилище записей журнала аудита FGA, уже существует в схеме `SYS`. Обычный аудит сначала необходимо включить на уровне базы данных, только потом можно будет регистрировать действия с отдельными объектами. Для этого следует задать параметр инициализации `AUDIT_TRAIL`. Параметр не является динамическим, поэтому после его установки придется перезапустить базу данных, чтобы он начал действовать.

Успех или неудача

Обычный аудит можно настроить так, чтобы он проводился вне зависимости от того, успешным или неудачным оказалось действие пользователя. FGA работает только для успешно выполненных операций.

Отключение/включение

FGA можно временно отключить, а затем включить. С обычным аудитом так поступать нельзя. Для приостановки обычного аудита следует использовать команду `NOAUDIT` для объекта. Если впоследствии вы захотите заново применить те же настройки аудита, что и раньше, вам придется их вспомнить, так как эта информация будет утеряна.

Выборочный аудит строк

FGA регистрирует доступ к данным каждый раз, когда пользователь выполняет оператор, вне зависимости от того, сколько строк

было изменено. Для обычного аудита можно указать, следует ли вносить записи в журнал по одной на обращение к данным или по одной на сеанс. Таким образом, при обычном аудите возможно уменьшение размера журнала аудита.

Таблицы базы данных или файлы операционной системы

Обычный аудит можно настроить так, чтобы журнал записывался в таблицы базы данных (`AUDIT_TRAIL=DB`) или в файлы операционной системы (`AUDIT_TRAIL=OS`). Последний вариант удобен тогда, когда предполагается обращение к журналам не администратора базы данных, а аудитора. Использование файлов операционной системы удобно также тем, что резервные копии журналов можно делать соответствующими средствами для файлов. В Windows, если `AUDIT_TRAIL=OS`, журналы аудита пишутся в журнал событий (Event Log), и доступ к ним осуществляется специальным образом. Использование операционной системы позволяет обеспечить целостность журналов аудита. В отличие от журналов обычного аудита, журналы FGA могут храниться только в таблице базы данных `FGA_LOG$`. В FGA вы можете создать модули обработки для записи в файлы, но так как такие модули пишутся от имени владельца программного обеспечения Oracle, они не защищены от возможных злоупотреблений со стороны администраторов баз данных.

Объекты по умолчанию

Обычный аудит может быть задан для объектов по умолчанию – объектов, которые еще только будут созданы. Например, команда `AUDIT UPDATE ON DEFAULT;` указывает, что база данных должна включить аудит для операции обновления всех таблиц, в том числе и тех, которые еще не созданы. При создании новой таблицы она автоматически помещается под контроль аудита для обновлений. FGA позволяет создание политик только для уже существующих таблиц и представлений.

Выборочные столбцы

Обычный аудит не обеспечивает возможности детального распознавания действий, таких как доступ к отдельному столбцу. FGA позволяет выборочно вести аудит для определенных столбцов, в результате чего журнал аудита имеет приемлемые размеры.

Запись текста SQL и переменных связывания

Мы уже говорили о том, что FGA записывает текст выполненных операторов SQL и значения переменных связывания. Это поведение по умолчанию, которое можно изменить, задав параметр `audit_trail` процедуры `ADD_POLICY` пакета `DBMS_FGA`. При обычном аудите для сбора таких сведений необходимо установить параметр инициализации базы данных `AUDIT_TRAIL` в значение `DB_EXTENDED` и перезагрузить базу данных, чтобы настройки вступили в силу.

Привилегии

Для выдачи команды включения обычного аудита пользователю необходимо обладать системной привилегией AUDIT SYSTEM или AUDIT ANY. Для работы с FGA пользователю необходима только привилегия EXECUTE на пакет DBMS_FGA.

Как видите, FGA значительно отличается от обычного аудита даже при том, что в версии Oracle 10*g* их журналы стали очень похожи друг на друга. Их сходство привело к тому, что в Oracle 10*g* новое представление DBA_COMMON_AUDIT_TRAIL отображает записи из обоих журналов аудита: для обычного аудита и детального аудита.

Пользователи, не зарегистрированные в базе данных

Уже не раз говорилось о том, что FGA регистрирует не только того, кто выполнил какое-то действие, но и то, что собственно произошло: изменившаяся строка, номер SCN на момент выполнения действия, терминал и многое другое. Конечно, наиболее важными являются сведения о пользователе, выполнившем отслеживаемую операцию. Они заносятся в столбец DB_USER представления DBA_FGA_AUDIT_TRAIL.

Однако в некоторых случаях имя пользователя базы данных не идентифицирует реального пользователя. В главе 5 при обсуждении безопасности на уровне строк вы видели, что некоторые типы архитектуры (например, трехзвенные веб-системы) устроены так, что для создания пула соединений с базой данных используется один идентификатор пользователя. Веб-пользователь подключается к серверу приложений, который в свою очередь подключается к пулу соединений, из которого пользователю выделяется одно соединение. После того как пользователь переходит в режим ожидания, это соединение может быть передано другому пользователю. Тем самым относительно небольшое количество сеансов базы данных может обслуживать большое число пользователей.

Однако такой сценарий создает проблему идентификации реальных пользователей базой данных. Для базы данных именем пользователя является разделяемый идентификатор, используемый пулом соединений, а не имя стоящего за ним реального пользователя. Таким образом столбец DB_USER записей аудита FGA содержит разделяемый идентификатор пользователя, который не может обеспечить настоящую контролируемость. Чтобы точно определить пользователя, выполнившего действие, необходимо идентифицировать реального отдельного пользователя внутри пула. Как это сделать?

Существует несколько способов. Большинство из них основывается на передаче дополнительной информации о реальном пользователе во вспомогательных элементах данных, соответствующих сеансу. Эти элементы заполняются клиентом и считываются базой данных при

сборе данных об изменениях. Важнейшими элементами являются идентификатор клиента и контекст приложения. Мы уже встречались с ними в главе 5, поэтому сейчас сразу приступим непосредственно к тому, как использовать эти элементы для детального аудита.

Идентификатор клиента

Начиная с версии Oracle9*i* символьная строка переменной длины может использоваться в качестве атрибута пользовательского сеанса. Это значение можно впоследствии извлечь в другой программе при помощи запроса к представлению V$SESSION и получить отличительную информацию о реальном пользователе. Предположим, что пользователь подключается к базе данных как DBUSER и выдает такой оператор:

```
BEGIN
   DBMS_SESSION.set_identifier ('REAL_USER');
END;
/
```

В результате в столбец CLIENT_IDENTIFIER представления V$SESSION попадет значение REAL_USER. Из другого сеанса можно будет запросить значение данного столбца.

```
SELECT client_identifier
  FROM v$session
 WHERE SID = sid;
```

Возвращается REAL_USER.

Данные сведения доступны не только в представлении V$SESSION; они также отображаются в журналах FGA:

```
SELECT client_id
  FROM dba_fga_audit_trail;
```

Также возвращается значение REAL_USER.

Если данное значение соответствует имени реального пользователя, это позволит обеспечить контролируемость для данного пользователя.

Существует несколько возможностей надежного и безопасного задания идентификатора клиента, которые рассматривались в главе 5.

Контексты приложения

Контексты приложения – это множества пар имя-значение, которые могут быть определены в сеансе посредством выполнения специально определенной хранимой процедуры. Они чаще всего используются для контроля доступа к ресурсам базы данных согласно правилам, зависящим от текущего пользователя (которые подробно рассматривались при обсуждении безопасности на уровне строк в главе 5). Как и идентификаторы клиента из предыдущего раздела, контексты приложения могут использоваться для передачи индивидуальных характеристик реального пользователя. Любой из способов (или их комбинация)

обеспечивает возможность идентификации пользователей, не зарегистрированных в базе данных.

Описание и примеры использования контекстов приложения приведены в главе 5.

Отладка FGA

Как и любая другая сложная функциональность Oracle нижнего уровня, FGA может время от времени работать неожиданным образом. В этом разделе будут описаны наиболее часто встречающиеся ошибки, связанные с FGA, и способы их обработки. Я не буду говорить о дефектах собственно FGA – опубликованных или неопубликованных, которые могут влиять на его поведение. Они обычно зависят от выбора платформы и имеют преходящий характер. Поэтому мы сосредоточимся на ошибках, возникающих при типичном использовании FGA.

При возникновении какой-то проблемы с FGA в каталоге, определенном параметром инициализации базы данных USER_DUMP_DEST, формируется файл трассировки. Этот файл содержит важную информацию о точном условии возникновения ошибки и предлагает подсказки для ее дальнейшей диагностики. Рассмотрим фрагмент такого файла трассировки:

```
/opt/app/oracle/admin/gridr/udump/gridr_ora_14424.trc
Oracle 10g Enterprise Edition Release 10.1.0.3.0 - 64bit Production
With the Partitioning, Oracle Label Security, OLAP and Data Mining options
ORACLE_HOME = /opt/app/oracle/product/10g_gridr
System name:    SunOS
Node name:      smiley2.proligence.com
Release:        5.9
Version:        Generic_117171-12
Machine:        sun4u
Instance name: gridr
Redo thread mounted by this instance: 1
Oracle process number: 54
Unix process pid: 14424, image: oraclegridr@smiley2.proligence.com

*** 2005-07-12 19:01:44.337
*** ACTION NAME:() 2005-07-12 19:01:44.288
*** MODULE NAME:(SQL*Plus) 2005-07-12 19:01:44.288
*** SERVICE NAME:(SYS$USERS) 2005-07-12 19:01:44.288
*** SESSION ID:(165.51503) 2005-07-12 19:01:44.288
FGA supports simple predicates only - error 28138
FGA Policy EMP_DML
```

Две последние (выделенные жирным шрифтом) строки сообщают два важных факта:

- Имя политики – в данном случае EMP_DML.

- Ошибка – в данном случае FGA поддерживает только простые предикаты.

Ошибка возникла, когда пользователь обновлял таблицу. Чтобы увидеть условие аудита, выполним такой оператор:

```
SQL> SELECT policy_text
  2    FROM dba_audit_policies
  3   WHERE policy_name = 'EMP_DML';

POLICY_TEXT
-------------------------------------------
SALARY >= 1500 or EMPID=304
```

Условие аудита в этой политике, `SALARY >= 1500 or EMPID=304`, включает в себя несколько условий и не является простым предикатом. Сообщение об ошибке подтверждает наши наблюдения. Решение чрезвычайно просто: следует пересоздать политику с простым предикатом, таким как `SALARY >= 1500` или `EMPID=304`, но не с обоими условиями одновременно. Политики со сложными предикатами работают для операторов `SELECT`, но не для операторов DML.

В некоторых случаях FGA генерирует ошибки «молча», то есть не информируя пользователя о возникновении ошибок. Рассмотрим следующий пример: модуль обработки (если он определен) завершается с ошибкой для некоторых строк. В этом случае пользователь не получает никаких данных об ошибке. Кроме того, строки, для которых модуль обработки не выполнился, не возвращаются в результате пользовательского запроса, а пользователь и не подозревает о том, что какие-то строки не возвращены. В подобном случае запись в журнале трассировки является единственным свидетельством возникновения ошибки. Приведем фрагмент файла трассировки:

```
*** 2005-07-12 17:52:07.590
*** ACTION NAME:() 2005-07-12 17:52:07.536
*** MODULE NAME:(SQL*Plus) 2005-07-12 17:52:07.536
*** SERVICE NAME:(SYS$USERS) 2005-07-12 17:52:07.536
*** SESSION ID:(165.50693) 2005-07-12 17:52:07.536
---------------------------------
Error during execution of handler in Fine Grained Auditing
Audit handler  : begin ARUP.ACCESS_FLAGGER(:sn, :on, :pl); end;
Error Number 1  : 604
Logon user     : SYSMAN
Object Schema: ARUP, Object Name: EMP, Policy Name: EMP_SEL
*** 2005-07-12 17:52:32.891
Error Number 2: 1438----------------------------------
Error during execution of handler in Fine Grained Auditing
Audit handler  : begin ARUP.ACCESS_FLAGGER(:sn, :on, :pl); end;
Error Number 1  : 604
Logon user     : SYSMAN
Object Schema: ARUP, Object Name: EMP, Policy Name: EMP_SEL
```

Фрагмент состоит из двух частей. Первая идет от начала и заканчивается второй строкой дат `*** 2005-07-12 17:52:32.891`, указывающей момент первого возникновения ошибки. Строки второй части отображают истинную ошибку. Обратите внимание на выделенную жирным шрифтом строку: `Error Number 2: 1438`. Эта ошибка является истинной причиной того, что операция не была выполнена успешно. Код ошибки можно найти при помощи утилиты `oerr`:

```
Smiley2:/opt/app/oracle/admin/gridr/udump>oerr ora 1438
01438, 00000, "value larger than specified precision allows for this column"
// *Cause:
// *Action:
```

Теперь причина ошибки ясна: модуль обработки пытался вставить в столбец значение, превышающее допустимое для данного столбца. Изменение значения приведет к исчезновению ошибки.

Последняя ошибка убеждает нас в следующем: если вы включаете детальный аудит для какой-то таблицы базы данных, обязательно просматривайте файлы трассировки (которые хранятся в каталоге, определенном параметром инициализации базы данных `USER_DUMP_DEST`) для отлавливания «безмолвных» ошибок. Возможно, вы и так уже просматриваете файлы трассировки, но теперь появляется дополнительная причина не забывать об этом.

FGA в двух словах

- FGA может регистрировать выборку данных из таблиц (`SELECT`) в Oracle9*i* и любые виды DML-операций в Oracle 10*g*. Записи детального аудита ведутся в таблице аудита `FGA_LOG$` схемы `SYS`.
- Для корректной работы FGA необходимо использовать оптимизатор по стоимости, в противном случае в таблицу аудита может попасть множество ложных записей.
- Детальный аудит ведет журнал в режиме автономных транзакций. Даже если операция DML завершается с ошибкой и откатывается, запись в журнале остается. Это также может приводить к появлению ложных записей.
- Журналы аудита отображают точный текст оператора SQL, выданного пользователем, значения переменных связывания (если они использовались), системный номер изменений (SCN) на момент запроса, а также различные атрибуты сеанса, такие как имя пользователя базы данных, имя пользователя операционной системы, временную метку и многое другое.

Заключение

Детальный аудит является замечательным инструментом регистрации пользовательских обращений к базе данных, сохраняя сведения, которые вы можете использовать для отслеживания пользовательских действий и обеспечения контролируемости. Можно настроить детальный аудит так, чтобы записывать обращения определенных пользователей к определенным таблицам и в определенных условиях (например, при упоминании конкретных столбцов или при выполнении операций в определенное время дня). Многие администраторы баз данных воспринимают FGA только как средство повышения безопасности за счет обеспечения контролируемости, но на самом деле вы также можете применять FGA для анализа способов доступа к данным, исследования использования операторов SQL и другой деятельности по повышению производительности базы данных. FGA позволяет избавиться или, по крайней мере, уменьшить зависимость от сложных триггерных механизмов, используемых многими администраторами баз данных для отслеживания и реагирования на обращения к базе данных.

7

Генерирование случайных значений

Базы данных предназначены для хранения достоверных фактов; мы тщательно собираем данные, касающиеся нашего бизнеса или другой деятельности, всеми силами защищаем их и поддерживаем в целости и сохранности. Зачем нам может понадобиться нечто совершенно противоположное – непредсказуемые случайные значения? Давайте рассмотрим несколько ситуаций, в которых администратору базы данных может потребоваться генератор случайных чисел:

- Вы создаете временные пароли или идентификаторы для зарегистрированных пользователей веб-сайта.
- Вы участвуете в разработке приложения, требующего тщательного всестороннего тестирования (производительности, масштабирования, точности и т. д.), и в рамках этого проекта вам необходимо создать набор тестов с представительными данными. Под *представительными* понимаются такие данные, которые каким-то образом соответствуют той сущности, которую они представляют; например, столбец с номерами счетов должен отражать формат реальных номеров, имя должно состоять из букв и т. п.
- Вы хотите протестировать эффект, создаваемый некоторым важным структурным компонентом базы данных, таким как индексы. Чтобы ответить на вопросы о количестве индексов, об индексируемых столбцах, о количестве собираемых в процессе анализа гистограмм, о целесообразности использования блочной выборки (block sampling) и тому подобных, необходимы большие массивы данных, отражающих реальные ситуации. Не всегда есть возможность получить эти данные со стороны, поэтому может возникнуть потребность в их создании. В таком случае важно сгенерировать такие данные, которые достаточно случайны, но в то же время соответствуют реальным образцам.

- Вы создаете инфраструктуру шифрования, и вам необходимо сгенерировать ключ. Oracle9*i* и последующие версии позволяют генерировать по-настоящему случайные ключи, в более ранних версиях вам придется делать это самостоятельно.

В этой главе рассказывается о том, как при помощи PL/SQL генерировать случайные значения (числовые и строковые), представляющие реальные значения. Рассматриваются способы задания начального значения, позволяющие получать случайные последовательности. Приведено множество примеров и фрагментов кода, ориентированных на создание тестовых данных (имен клиентов и сумм на их счетах) для банковского приложения, которые вы легко сможете использовать в собственных разработках.

Большинство примеров этой главы основано на использовании встроенного пакета Oracle `DBMS_RANDOM`. В этом пакете имеются две ключевые функции, возвращающие случайные значения:

Для чисел

Функция `VALUE` возвращает положительное число с плавающей точкой в диапазоне от 0 до 1 с точностью 38 знаков, например 0,034869472.

Для строк

Функция `STRING` возвращает случайную строку символов с указанными пользователем длиной и распределением символов.

Использование этих и других функций пакета `DBMS_RANDOM` будет проиллюстрировано в примерах этой главы.

Генерирование случайных чисел

Одно из наиболее распространенных применений генераторов случайных значений заключается в создании случайных чисел. Числа могут быть разными: положительными или отрицательными, целыми или дробными и т. п. Давайте начнем с простейшей разновидности – положительных чисел.

Генерирование положительных чисел

Требуется сгенерировать ряд чисел, представляющих значения остатков на счетах для наших тестовых данных. Остаток на счете может быть представлен целочисленным значением или значением с плавающей точкой с двумя разрядами в дробной части, например 12345,98. Примем для простоты, что банк не допускает перерасхода – то есть баланс не может быть отрицательным. Функция `VALUE` пакета `DBMS_RANDOM` возвращает положительное число большее или равное нулю и меньшее 1, имеющее 38 знаков после запятой. Вот как эта функция используется для получения числа:

```
CREATE OR REPLACE FUNCTION get_num
   RETURN NUMBER
IS
   l_ret   NUMBER;
BEGIN
   l_ret := DBMS_RANDOM.VALUE;
   RETURN l_ret;
END;
```

Будучи вызванной, функция возвращает число такого вида:

```
SQL> COLUMN get_num FORMAT 0.99999999999999999999999999999999999999

SQL> SELECT get_num FROM dual;

                                GET_NUM
---------------------------------------
.46175121313611771875955981218546375555
```

Благодаря тому что функция имеет степень чистоты WNDS (write no database state – не менять состояние данных), ее можно вызывать прямо в операторе SELECT или внутри INSERT.

Отвлечемся ненадолго и рассмотрим приложение другого типа. Предположим, что необходимо написать программу для научных расчетов, которая считывает показания термометра и сохраняет их для последующего анализа. Набор чисел можно сформировать в таком коде:

```
BEGIN
   FOR ctr IN 1 .. 10
   LOOP
      DBMS_OUTPUT.put_line ('Temperature = ' || get_num || ' Celsius');
   END LOOP;
END;
```

Получим:

```
Temperature = .72457630554564072922992665224928198851 Celsius
Temperature = .45153898247979596526993387440338586133 Celsius
Temperature = .59553753239053135052219502299103821094 Celsius
Temperature = .04579937433263277972474686173668834983 Celsius
Temperature = .16737654763541403045022512399535308709 Celsius
Temperature = .41418437394466751017150096580834486541 Celsius
Temperature = .66938830783835508007657316426304773073 Celsius
Temperature = .54439158666311298674126067789688453747 Celsius
Temperature = .01701471024660024158283343878008518868 Celsius
Temperature = .78426530713204463490011654216600338328 Celsius
```

Обратите внимание, что значения температуры совершенно случайны. Теперь представим, что их надо сохранить для анализа в будущем. Используем следующий код для записи в таблицу TEMPERATURES, имеющую такой вид:

```
SQL> DESC temperatures
 Name                 Null?    Type
```

```
------------------- -------- -------

READING_ID                   NUMBER
TEMPERATURE                  NUMBER
```

Вот программа, записывающая случайные значения в эту таблицу:

```
BEGIN
   FOR ctr IN 1 .. 10
   LOOP
      INSERT INTO temperatures
           VALUES (ctr, get_num);
   END LOOP;
END;
```

Сделав выборку из таблицы, увидим полученные значения.

```
SQL> COLUMN temperature FORMAT 0.99999999999999999999999999999999999999

SQL> SELECT *
  2> FROM temperatures;

READING_ID                              TEMPERATURE
---------- ------------------------------------
         1 .667338602247281954526409542630693215 6
         2 .354509992258827837872055900075108815 9
         3 .089242570815172346785112867801556047 4
         4 .159997918646677274523012508947348303 2
         5 .429123979218669780845026573976545568 6
         6 .012391831990894964321724906312383977 5
         7 .192638609831779499063694365444124767 9
         8 .963138345202001606154334058632478286 9
         9 .767860198517120118062644588247325879 4
        10 .094493424805659411268321447633421126 3
```

Этот пример можно расширить, увеличив количество записываемых показаний, добавив другие источники данных (дополнительные термометры) и т. п.

Задание точности

В предыдущем примере генерировались числа с плавающей точкой с 38 знаками после запятой. Числа такого вида не очень подходят для приложения, работающего с номерами банковских счетов, которые должны иметь целочисленный формат и, как правило, фиксированное количество разрядов. Нет проблем! Изменим нашу функцию, чтобы сделать ее более полезной. Например, в следующем варианте функции возвращаемое случайное число находится в диапазоне от 1 до 100:

```
CREATE OR REPLACE FUNCTION get_num
   RETURN PLS_INTEGER
IS
   l_ret   PLS_INTEGER;
BEGIN
```

```
   l_ret := ROUND (100 * DBMS_RANDOM.VALUE);
   RETURN l_ret;
END;
```

Увеличивая значение множителя, можно повысить разрядность получаемых случайных чисел.

Тот же принцип применим к генерированию чисел с плавающей запятой. В США значения денежных сумм принято хранить в долларах и центах. Цент равен одной сотой части доллара, следовательно, требует двух знаков после запятой. Под этот формат подходят валюты многих стран. В этом случае генерируемое случайное значение остатка должно иметь два знака в дробной части. Простейший способ достичь этого заключается в задании масштаба в функции ROUND. Добавим в нашу функцию параметр, чтобы получить более общее решение:

```
CREATE OR REPLACE FUNCTION get_num (
   p_precision   IN   PLS_INTEGER,
   p_scale       IN   PLS_INTEGER := 0
)
   RETURN NUMBER
IS
   l_ret   NUMBER;
BEGIN
   l_ret := ROUND (10 * p_precision * DBMS_RANDOM.VALUE, p_scale);
   RETURN l_ret;
END;
```

С этими аргументами легко можно получать случайные числа нужного формата.

Задание диапазона

Предположим, что нужно сгенерировать суммы остатков, попадающие в определенный диапазон. К примеру, банк может требовать некоторой минимальной суммы на счете, поэтому случайные значения должны быть больше допустимого минимума и, вероятно, меньше некоторого разумного максимума. Для выполнения этого требования перегруженная функция VALUE в пакете DBMS_RANDOM принимает два параметра, low и high, и возвращает значение из этого диапазона. Вот пример вызова перегруженной функции для получения случайных чисел в диапазоне от 1000 до 9999:

```
l_ret := DBMS_RANDOM.VALUE(1000,9999)
```

Длинная дробная часть сгенерированного случайного числа может выглядеть устрашающе, но от нее можно избавиться с помощью функции ROUND. Изменим функцию генерации остатков так, чтобы получать значения в заданном диапазоне:

```
/* Файл на веб-сайте: get_num_1.sql */
CREATE OR REPLACE FUNCTION get_num (
```

```
   p_highval   NUMBER,
   p_lowval    NUMBER := 0,
   p_scale     PLS_INTEGER := 0
)
   RETURN NUMBER
IS
   l_ret   NUMBER;
BEGIN
   l_ret := ROUND (DBMS_RANDOM.VALUE (p_lowval, p_highval), p_scale);
   RETURN l_ret;
END;
```

Заметьте, эта функция достаточно универсальна и удовлетворяет многим задачам по генерированию случайных чисел, таким как:

- Генерация номеров банковских счетов, которые представляются десятизначными целыми числами. Эти номера строго десятизначные, т. е. не могут содержать ведущие нули.

```
SQL> EXEC DBMS_OUTPUT.put_line(get_num(1000000000,9999999999))

2374929832
```

- Использование той же функции для генерации значений остатков на счетах, которые имеют два знака после запятой и находятся в диапазоне от 1000 (требуемый банком минимальный остаток) до 1 000 000 (предполагаемое максимальное значение остатка). Для генерации двух дробных разрядов параметру `p_scale` требуется передать значение 2.

```
SQL> EXEC DBMS_OUTPUT.put_line(get_num(1000,1000000,2))

178861.81
```

- Использование функции в научных приложениях, например при считывании показаний высокочувствительного термометра, применяемого в атомных установках, где значения представлены тремя разрядами до и десятью разрядами после запятой (то есть числа находятся в диапазоне от 100,0000000000 до 999,9999999999):

```
BEGIN
   DBMS_OUTPUT.put_line (
      get_num(100,999.9999999999,10));
END;
/
607.1872599141
```

Генерирование отрицательных чисел

До сих пор мы рассматривали только генерирование положительных чисел и не касались отрицательных. Но предположим, что наше банковское приложение позволяет работать с перерасходом средств, то есть с временным заимствованием денег со счетов. В таком случае ос-

таток на счете может стать отрицательным (состояние, известное как «овердрафт»). В тестовых данных могут понадобиться и такие счета. Функция DBMS_RANDOM.RANDOM позволяет сделать это в PL/SQL.

Функция RANDOM

В пакете DBMS_RANDOM имеется функция RANDOM для генерации случайных целых чисел, вызываемая без параметров и возвращающая двоичное целое в диапазоне от -2^{31} до 2^{31}, то есть целое десятизначное десятичное число. Используется она просто:

```
DECLARE
   l_ret   NUMBER;
BEGIN
   l_ret := DBMS_RANDOM.random;
   DBMS_OUTPUT.put_line ('The number generated = ' || l_ret);
   l_ret := DBMS_RANDOM.random;
   DBMS_OUTPUT.put_line ('The number generated = ' || l_ret);
   l_ret := DBMS_RANDOM.random;
   DBMS_OUTPUT.put_line ('The number generated = ' || l_ret);
END;
/
```

В результате получим:

```
The number generated = 865225855
The number generated = 1019041205
The number generated = -1410740185
```

Обратите внимание, что *в данном конкретном примере* присутствуют одновременно отрицательные и положительные целые числа длиной до 10 разрядов. Нет никакой гарантии, что при вызове этой программы вы *получите* отрицательное случайное число. Если необходимо, чтобы получаемое отрицательное случайное число соответствовало определенному формату, используйте комбинацию функций RANDOM и VALUE, как показано ниже.

Функцию get_num можно изменить так, чтобы она принимала параметр, позволяющий ей генерировать отрицательные числа.

```
/* Файл на веб-сайте: get_num_2.sql */
CREATE OR REPLACE FUNCTION get_num (
   p_highval             NUMBER,
   p_lowval              NUMBER := 0,
   p_negatives_allowed   BOOLEAN := FALSE,
   p_scale               PLS_INTEGER := 0
)
   RETURN NUMBER
IS
   l_ret   NUMBER;
   l_sign  NUMBER := 1;
BEGIN
   IF (p_negatives_allowed)
```

```
    THEN
       l_sign := SIGN (DBMS_RANDOM.random);
    END IF;

    l_ret := l_sign *
             ROUND (DBMS_RANDOM.VALUE (p_lowval, p_highval)
                  , p_scale);
    RETURN l_ret;
END;
```

Здесь сохранена функциональность исходной версии функции `get_num` (принимающей параметры диапазона `high` и `low` и параметр точности `scale`) и добавлена новая возможность, позволяющая генерировать как положительные, так и отрицательные случайные числа.

Задание начального значения для генератора случайных чисел

Что делает случайные числа действительно случайными? Чтобы вы поняли, о чем идет речь, вот вам упражнение: быстро придумайте число, *любое* число. Какое число пришло вам на ум? Наверняка оно что-нибудь значит для вас: ваш возраст, часть вашего телефонного номера или номер дома – что-то, что вам знакомо или находится в поле зрения в данный момент. Другими словами, выбор этого числа не был случайным, а, следовательно, может быть повторен или предсказан.

Хорошо, скажете вы. Вам нужно случайное число? Я закрою глаза и буду беспорядочно давить на клавиши калькулятора. Нет сомнений, что *такое* число будет случайным.

Скорее всего, нет. Если вы внимательно посмотрите на такое число, то заметите, что некоторые цифры образуют повторяющиеся последовательности. Это естественно: если вы нажали клавишу (например, 9), то высока вероятность того, что вы нажмете ее еще раз, так как ваш палец завис в непосредственной близости от нее.

Надеюсь, после этих мысленных экспериментов вы понимаете, что человеку достаточно трудно придумать действительно случайное число. Так же и машина не может просто взять число «из ниоткуда». Есть ряд математических методов, обеспечивающих истинную случайность, но их рассмотрение не входит в задачи этой книги. Во всех генераторах случайных чисел есть общий, повсеместно доступный источник «случайностей», такой как системный таймер, который гарантирует, что значения, взятые в любые два момента времени, будут различны. Этот случайный компонент, концептуально аналогичный области перемещения вашего пальца над клавиатурой калькулятора при наборе случайного числа, называется *начальным значением последовательности случайных чисел* (*seed*). Если такое начальное значение выбрано достаточно случайно, то и генерируемое число действительно будет случайным. Если начальное значение не меняется, генерируемое чис-

ло может не быть в полной мере случайным. Образец может повториться и быть легко угадан. Следовательно, выбор начального значения последовательности очень важен при генерировании случайных чисел.

Пакет DBMS_RANDOM включает в себя процедуру INITIALIZE, которая предоставляет начальное значение последовательности случайных чисел, используемое затем в функции-генераторе случайных чисел. Вот пример вызова этой программы:

```
BEGIN
   DBMS_RANDOM.initialize (10956782);
END;
```

Процедура принимает аргумент типа BINARY_INTEGER и использует его в качестве начального значения последовательности случайных чисел. Чем длиннее такое начальное значение, тем более случайным будет генерируемое число. Как правило, значения, содержащие более 5 знаков, обеспечивают приемлемую степень случайности.

Совершенно не обязательно явно вызывать процедуру INITIALIZE; при использовании пакета DBMS_RANDOM это вообще не требуется. По умолчанию, если не было явной инициализации, Oracle выполняет автоматическую инициализацию значениями даты и идентификаторов пользователя и процесса.

Итак, для достижения истинной случайности, значение не должно быть постоянным. Хорошим источником данных для получения начального значения последовательности случайных чисел является системное время, которое гарантированно принимает различные значения на протяжении 24-часового периода, например:

```
BEGIN
   DBMS_RANDOM.initialize (
      TO_NUMBER (TO_CHAR (SYSDATE, 'MMDDHHMISS')));
END;
```

Здесь в качестве начального значения использованы месяц, день, час, минуты и секунды. Это значение используется при первой генерации случайного числа, однако, если использовать это же значение на протяжении всего сеанса, случайность может быть нарушена. Следовательно, необходимо периодически задавать новое начальное значение. Для этого существует процедура SEED, которая, как и INITIALIZE, принимает параметр типа BINARY_INTEGER.

```
BEGIN
   DBMS_RANDOM.seed (TO_CHAR (SYSDATE, 'mmddhhmiss'));
END;
```

Когда генерирование начальных значений в рамках сеанса больше не требуется, можно вызвать процедуру TERMINATE, как показано ниже, чтобы прекратить создание новых значений (и перестать расходовать на это циклы ЦПУ).

```
BEGIN
   DBMS_RANDOM.terminate;
END;
```

Генерирование строк

В предыдущих разделах рассмотрены основы генерирования случайных значений, соответствующие функции и их инициализация, но все приведенные примеры относились к генерированию чисел. В этом разделе мы рассмотрим генерирование случайных символьных строк с помощью функции STRING пакета DBMS_RANDOM. Функция STRING принимает два параметра, opt и len.

Первый из них, opt, определяет тип генерируемой строки. Его возможные значения перечислены в табл. 7.1.

Таблица 7.1. Возможные значения параметра opt

Значение opt	Действие
u	Генерировать только буквы в верхнем регистре (например, DFTHNDSW)
l	Генерировать только буквы в нижнем регистре (например, pikdcdsd)
a	Генерировать только буквы в обоих регистрах (например, DeCWCass)
x	Генерировать буквы и цифры в верхнем регистре (например, A1W56RTY)
p	Генерировать любые печатные символы (например, $\$2sw&*)

Второй параметр, len, определяет длину генерируемой символьной строки. Он необходим, если приложению требуются случайные строки определенной длины. Создадим первую нашу функцию генерации случайных строк:

```
CREATE OR REPLACE FUNCTION get_random_string (
   p_len    IN   NUMBER,
   p_type   IN   VARCHAR2 := 'a'
)
   RETURN VARCHAR2
AS
   l_retval   VARCHAR2 (200);
BEGIN
   l_retval := DBMS_RANDOM.STRING (p_type, p_len);
   RETURN l_retval;
END;
/
```

Создание 40-символьной строки из букв в смешанном регистре с помощью нашей новой функции выглядит так:

```
SQL> EXEC DBMS_OUTPUT.put_line(get_random_string(40))
XaCbNwzpkGEsgqzCCdEykCycEtLlvoMOxrPnwanj

PL/SQL procedure successfully completed.

SQL> EXEC DBMS_OUTPUT.put_line(get_random_string(40))
hUctRtmsTWsedxqcTNNIlMDhyTgcQmmkyhrCwkUY

PL/SQL procedure successfully completed.
```

Как показано в таблице, задавая различные значения параметра p_opt, можно создавать случайные строки различных типов, состоящие, в частности, только из букв в верхнем регистре, только из букв в нижнем регистре, из букв и цифр, из любых печатных символов.

Как и для чисел, для генерации строк можно указать начальное значение последовательности с помощью той же процедуры SEED, имеющей перегруженную версию, принимающую аргумент типа VARCHAR2. В качестве начального значения можно, как и раньше, использовать системное время, преобразованное к символьному типу.

```
BEGIN
   DBMS_RANDOM.seed (TO_CHAR (SYSDATE, 'mmddhhmiss'));
END;
```

Давайте вспомним о цели, поставленной в начале этой главы: мы хотели создать тестовые данные, содержащие имена клиентов и значения остатков на их счетах, используя надежный способ рандомизации, с тем, чтобы полученные значения номеров и сумм были действительно случайными. Мы обсудили способ создания сумм остатков, а что насчет имен?

Можно было бы использовать нашу функцию get_random_string, но сейчас она возвращает только строки заданной длины. В действительности имена имеют переменную длину. Чтобы сгенерировать достаточно разнообразный набор имен, усовершенствуем нашу функцию: будем задавать минимальную и максимальную длину, а фактическая длина будет случайной в рамках этого диапазона. Вот вторая версия функции генерации случайных строк:

```
CREATE OR REPLACE FUNCTION get_random_string (
   p_minlen   IN   NUMBER,
   p_maxlen   IN   NUMBER,
   p_type     IN   VARCHAR2 := 'a'
)
   RETURN VARCHAR2
AS
   l_retval   VARCHAR2 (200);
BEGIN
   l_retval :=
       DBMS_RANDOM.STRING (p_type
                         , DBMS_RANDOM.VALUE (p_minlen, p_maxlen));
   RETURN l_retval;
END;
```

Вызывая функцию DBMS_RANDOM.STRING, передаем ей в качестве параметра длины строки случайное число из указанного параметрами диапазона. Проверим, как это работает:

```
SQL> EXEC DBMS_OUTPUT.put_line(get_random_string(10,20))
DPZmDqQFcwhnqRL

PL/SQL procedure successfully completed.

SQL> EXEC DBMS_OUTPUT.put_line(get_random_string(10,20))
goJVifYdSeQ

PL/SQL procedure successfully completed.
```

Обратите внимание, что получены строки разной длины. Такой подход позволяет генерировать случайные строки для самых разных приложений, будь то имена клиентов, идентификаторы пользователей вебсайта или временные пароли. Однако, как станет понятно далее, наиболее полезной представляется возможность генерирования начальных значений последовательностей и ключей для шифрования.

Давайте напишем программу заполнения записей счетов случайными данными. Таблица ACCOUNTS выглядит так:

```
SQL> DESC accounts
 Name                 Null?    Type
 -------------------- -------- -------

 ACCOUNT_NO                    NUMBER
 BALANCE                       NUMBER(9,4)
 ACCOUNT_NAME                  VARCHAR2(20)
```

Вот программа, заполняющая ее случайными значениями:

```
DECLARE
   i   NUMBER;
BEGIN
   FOR i IN 1 .. 10
   LOOP
      DBMS_RANDOM.seed (TO_NUMBER (TO_CHAR (SYSDATE, 'hhmiss')));

      INSERT INTO accounts (account_no, balance, account_name)
           VALUES (i
                 , DBMS_RANDOM.VALUE (10000, 99999)
                 , DBMS_RANDOM.STRING ('x', DBMS_RANDOM.VALUE (10, 20)));
   END LOOP;
END;
/
```

После выполнения этого PL/SQL-блока мы увидим записи такого вида:

```
SQL> SELECT * FROM accounts;

ACCOUNT_NO    BALANCE ACCOUNT_NAME
---------- ---------- --------------------
         1 91772.2043 FZIGIONR8OU
         2 91772.2043 FZIGIONR8OU
         3 91772.2043 FZIGIONR8OU
```

```
         4 91772.2043 FZIGIONR8OU
         5 91772.2043 FZIGIONR8OU
         6 91772.2043 FZIGIONR8OU
         7 91772.2043 FZIGIONR8OU
         8 91772.2043 FZIGIONR8OU
         9 86258.8032 65YJYV9NMMBGMRP5CMP7
        10 86258.8032 65YJYV9NMMBGMRP5CMP7
```

Как видите, результат весьма далек от ожидаемого. Слишком много совпадающих значений сумм и имен. Это совсем не похоже на по-настоящему случайный набор данных. В чем же ошибка?

Проблема заключается в начальном значении последовательности. Обратите внимание, что в 6 строке фрагмента кода функции `seed` передается значение, составленное из часов, минут и секунд. Но этот код выполняется менее чем за секунду, из-за чего каждый раз устанавливается *одно и то же* начальное значение, поэтому и сгенерированные случайные значения оказываются одинаковыми. В приведенном примере в итерациях с 1 по 8 использовались одинаковые начальные значения, вследствие чего и все «случайные» значения оказались одинаковыми. Начальное значение последовательности *изменилось* в девятой итерации, и это привело к изменению «случайного» значения.

Решить эту проблему можно простым удалением из программы строки с вызовом функции `seed`. Вот измененный код с закомментированной строкой:

```
DECLARE
   i   NUMBER;
BEGIN
   FOR i IN 1 .. 10
   LOOP
      -- DBMS_RANDOM.seed (TO_NUMBER (TO_CHAR (SYSDATE, 'hhmiss')));

      INSERT INTO accounts
           VALUES (i, DBMS_RANDOM.VALUE (10000, 99999),
                   DBMS_RANDOM.STRING ('x', DBMS_RANDOM.VALUE (10, 20)));
   END LOOP;
END;
/
```

Теперь, посмотрев на полученные значения, увидим действительно случайный набор данных.

```
SQL> SELECT * FROM accounts;

ACCOUNT_NO    BALANCE ACCOUNT_NAME
---------- ---------- --------------------
         1 18344.0416 ELR8PWCSIAPKF1POH
         2  94702.904 XQGBFVQGQI8QPAV6NIN
         3 64261.8317 POF99DU2DAHOXE5AC7
         4 95369.3182 42IT6V7XOAF7
         5 65451.8237 6ZU5H91XEGMVO
         6 68695.4939 4VNP4KWJN6Y
```

```
 7 71474.0692 9OLNOSJNKE5CO
 8 78402.0396 IG9Z3KEFZ35YCXIER9N
 9 63726.3395 86JK18HJEON
10 12416.5512 JRQC39C5KOLA
```

Отнеситесь внимательно к полученному уроку. Использование *одинаковых* начальных значений последовательности случайных чисел приводит к предсказуемости результата работы генератора. Не задавая начального значения, мы получим действительно случайные числа. В примере выше программа выполнялась слишком быстро, опережая изменения начального значения. Но что делать, если требуется получать случайные значения, мы хотим использовать некоторое начальное значение, но ничего не знаем о скорости выполнения программы?

Решение заключается в добавлении к начальному значению дополнительного (и *гарантированно* изменяющегося) элемента. В приведенном фрагменте этим элементом может быть переменная цикла `i`, которая обязательно изменяет свое значение внутри цикла. Тогда код можно переписать следующим образом:

```
 1  DECLARE
 2     i   NUMBER;
 3  BEGIN
 4     FOR i IN 1 .. 10
 5     LOOP
 6        DBMS_RANDOM.seed (i || TO_NUMBER (TO_CHAR (SYSDATE, 'hhmiss')));
 7
 8        INSERT INTO accounts
 9             VALUES (i, DBMS_RANDOM.VALUE (10000, 99999),
10                     DBMS_RANDOM.STRING ('x', DBMS_RANDOM.VALUE (10, 20)));
11     END LOOP;
12* END;
```

Обратите внимание на строку 6, где теперь к начальному значению добавляется компонент `i`. Это гарантирует, что значение каждый раз будет новым, даже при очень быстром выполнении программы. Такой подход можно использовать в любых программах, использующих случайные значения.

Никогда не устанавливайте начальное значение последовательности равным тому, каким оно было до вызова функции генерации случайного значения. Если вы в качестве начального значения используете некую переменную, значение которой потенциально может остаться неизменным (такую как текущее время), добавьте к ней другую переменную, которая гарантированно изменяется (например, счетчик цикла).

Проверка на случайность

Получив, как описано в предыдущем разделе, случайные значения, вы, возможно, захотите убедиться в том, что они *действительно* явля-

ются случайными. Для этого проверьте статистическое распределение случайных данных в таблице.

```
SQL> SELECT MIN (balance), MAX (balance)
  1       , AVG (balance), STDDEV (balance)
  2    FROM accounts;

MIN(BALANCE) MAX(BALANCE) AVG(BALANCE) STDDEV(BALANCE)
------------ ------------ ------------ ---------------
    10008.03     99889.97   54948.4654      25989.9271
```

В данном случае среднее значение (которое в статистике также часто называют *математическим ожиданием*) баланса составляет 54948,4654 при среднеквадратичном отклонении 25989,9271. Согласно законам математической статистики, это означает следующее распределение значений в таблице:

Пусть A – среднее, а S – среднеквадратичное отклонение, тогда:

- Приблизительно 68% значений находятся в интервале от (A–S) до (A+S)
- Приблизительно 95% значений находятся в интервале от (A–2×S) до (A+2×S)
- Приблизительно 99.7% значений находятся в интервале от (A–3×S) до (A + 3×S)

Если распределение имеет такой вид, то его называют *нормальным распределением*. Однако хотелось бы получить равномерное, а *не* нормальное распределение. Вот наши значения:

A = 54948,4654; S = 25989,9271

Следовательно, 68% данных лежат в диапазоне от 28958,5383 до 80938,3925, согласно следующим выражениям:

54948,4654 – 25989,9271 = 28958,5383

и

54948,4654 + 25989,9271 = 80938,3925

Эти значения показывают, что значения списка весьма разнообразны и не очень скучены вокруг среднего значения. Так что он удовлетворяет определению истинно случайной выборки. При создании тестового стенда для проверки предположений вам придется построить несколько случайных выборок данных, и исследования, подобные только что проведенным нами, помогут вам убедиться в случайности выборок. Мы еще вернемся к этой теме в следующем разделе.

Следование статистическим шаблонам

Предположим, что вы создаете таблицу заказчиков и заполняете ее именами, которые впоследствии будут использоваться при поиске. Как сгенерировать такие значения? Например, должны ли значения быть произвольными наборами символов, как предложенные ниже?

```
-32nr -32nr3121ne -e21e
323-=11r- r
0-vmdw-dwv0-[o- rr0-32r2 0
r4i32r -rm32r3p=x ewifef-432fr32o3-==
```

Я получил эти значения, случайным образом нажимая на кнопки клавиатуры. Но они вряд смогут служить удачными примерами имен заказчиков. В конце концов, имена представляют собой некие определенные последовательности символов (как Jim или Jane), и при создании таблицы тестовых данных необходимо обеспечить максимальную похожесть тестовых значений на реальные. Ваши значения должны браться из множества известных значений, но случайным образом, а не по какому-то предопределенному правилу.

Еще одним важным аспектом, который необходимо учитывать, является распределение. Уникальных имен очень мало. Например, в США часто встречаются такие мужские имена, как John, James или Scott (но очень редко такие, как Arup). Так что при формировании тестовых данных необходимо, чтобы распределение было случайным, но при этом отражало и реальную статистическую модель. Например, будем считать, что среди нашего населения имена распределяются следующим образом:

10%	Alan
10%	Barbara
5%	Charles
5%	David
15%	Ellen
20%	Frank
10%	George
5%	Hillary
10%	Iris
10%	Josh

Если заполнить столбец FIRST_NAME таблицы ACCOUNTS согласно этому шаблону, то для 10% записей значением FIRST_NAME будет Alan, для других 10% – Barbara и т. д. Теперь давайте попытаемся заполнить остальные столбцы таким же образом, чтобы они отражали реальное положение вещей. Посмотрим на перечень столбцов измененной таблицы ACCOUNTS.

```
SQL> DESC accounts
 Name              Null?    Type
 ----------------- -------- ------------
 ACC_NO            NOT NULL NUMBER
 FIRST_NAME        NOT NULL VARCHAR2(30)
 LAST_NAME         NOT NULL VARCHAR2(30)
```

```
ACC_TYPE          NOT NULL VARCHAR2(1)
FOLIO_ID                   NUMBER
SUB_ACC_TYPE               VARCHAR2(30)
ACC_OPEN_DT       NOT NULL DATE
ACC_MOD_DT                 DATE
ACC_MGR_ID                 NUMBER
```

Предположим, что для того чтобы сделать распределение данных приближенным к реальному, я буду создавать данные для столбцов по шаблонам, приведенным в таблице.

Имя столбца	Описание	Шаблон данных
ACC_NO	Номер счета	Любое число, не более 10 разрядов
FIRST_NAME	Имя	10% – Alan 10% – Barbara 5% – Charles 5% – David 15% – Ellen 20% – Frank 10% – George 5% – Hillary 10% – Iris 10% – Josh
LAST_NAME	Фамилия	Любая буквенная строка длиной от 4 до 30 символов, при этом 25% значений – «Smith»
ACC_TYPE	Тип счета: сберегательный (savings), текущий (checking) и т. д.	20% на каждый из S, C, M, D и X
FOLIO_ID	Идентификатор листа из других систем	Половина – значения NULL, вторая половина – номер, связанный с номером счета
SUB_ACC_TYPE	Для клиентов-юридических лиц номера суб-счетов (если они есть)	75% – значения NULL. Из оставшихся: 5% – S 20% – C
ACC_OPEN_DT	Дата открытия счета	Дата в диапазоне от текущей и до 500 дней назад
ACC_MGR_ID	Идентификатор администратора данного счета	Существует 5 администраторов, счета между которыми распределены в следующем процентном отношении: 1 – 40% 2 – 10% 3 – 10% 4 – 10% 5 – 30%

Как видите, требования являются умеренно сложными и в точности отражают распределение данных в реальной базе данных. Реальные клиенты будут иметь такие имена, как «Josh» и «Ellen», а не «XepqjEuF», поэтому имена следует выбирать из множества существующих имен. Фамилии в США встречаются самые разнообразные. Поэтому я выбрал полупроизвольное распределение, выделив 25% на чрезвычайно популярную фамилию «Smith». Остальные фамилии могут быть случайными.

Как же сгенерировать данные, отвечающие такому сложному набору правил? Я позаимствую пару страниц из учебника по теории вероятности и последую за теми авторами, которые любят использовать *метод Монте-Карло*. Следуя этому методу можно сгенерировать случайное число в диапазоне от 1 до 100 (включительно). Если промежуток времени достаточно велик, то вероятность генерирования какого-то конкретного числа (например, 6) будет равняться 0,01 или 1%. На самом деле вероятность генерирования любого числа равна 0,01. Если использовать те же рассуждения, то вероятность получения любого из двух чисел (например, 1 или 2), будет равняться 2%, а любого из трех чисел – 3%. Наконец, вероятность получения одного из чисел между 1 и 10 равна 10%. Используем эти знания для задания вероятности генерируемых случайных значений.

Генерирование строк

Обратимся, например, к значениям столбца ACC_TYPE, в котором с одинаковой вероятностью должны встречаться S, C, M, D и X, то есть вероятность каждого – 20%. Если генерировать целое число в промежутке от 1 до 5 (включительно), то вероятность появления каждого значения будет равна 20%. Затем используем SQL-функцию DECODE для получения значения ACC_TYPE на основе полученного числа.

```
1  SELECT DECODE (FLOOR (DBMS_RANDOM.VALUE (1, 6)),
2                 1, 'S',
3                 2, 'C',
4                 3, 'M',
5                 4, 'D',
6                 'X'
7                )
8*   FROM DUAL;
```

Давайте посмотрим, что происходит. Сначала генерируется число от 1 до 5 (строка 1). Генерируемое число меньше максимального значения, передаваемого в параметре, поэтому я указал число 6. Число должно быть целым, поэтому в строке 1 использована функция FLOOR, которая отсекает дробную часть числа. Применяя к сгенерированному числу функцию DECODE, получаем одно из значений: S, C, M, D или X. Вероятность генерирования любого из чисел 1, 2, 3, 4 и 5 равна 20%, соответственно так же будут распределены и буквы: по 20% на каждую.

Этот способ удобен для генерирования случайных, но предопределенных значений (как в нашем примере). Так же можно получить практически любые предопределенные случайные значения.

Генерирование случайных значений с NULL

Если вы вернетесь к требованиям для столбцов нашей таблицы, то заметите, что столбец FOLIO_ID несколько отличается от других. В нем должно быть заполнено только 50% значений, остальные должны содержать NULL. Как этого добиться?

Будем следовать тем же путем, что и в прошлый раз, добавив небольшую хитрость: проверку логического условия («да» или «нет»). При генерировании случайных чисел в диапазоне от 1 до 100 вероятность получения любого числа равна 1%. Следовательно, вероятность получения числа, меньшего, чем 51, составит ровно 50%. Используем этот факт в операторе CASE для получения значения.

```
SQL> SELECT CASE
  2            WHEN DBMS_RANDOM.VALUE (1, 100) < 51
  3               THEN NULL
  4            ELSE FLOOR (DBMS_RANDOM.VALUE (1, 100))
  5         END
  6* FROM DUAL;
```

В строке 2 проверяется, меньше ли полученное число, чем 51. Если меньше, то возвращаем NULL. Вероятность генерирования числа, меньшего, чем 51, равна 50%, поэтому мы получим 50% значений NULL. На другие 50% приходятся случайные числа в диапазоне от 1 до 100, которые можно использовать как FOLIO_ID.

Генерирование случайных строк случайной длины

Функция DMBS_RANDOM.STRING генерирует случайную строку, но фиксированной длины. Однако в реальности фамилии людей имеют *переменную* длину. Для столбца фамилий определено следующее требование: длина значений должна быть в промежутке между 4 и 30 символами. Для выполнения этого требования можно передавать функции STRING в качестве длины случайную величину, как мы это и сделаем в строке 6.

```
 1  BEGIN
 2     FOR i IN 1 .. 10
 3     LOOP
 4        DBMS_OUTPUT.put_line (    'Random String='
 5                    || DBMS_RANDOM.STRING ('A'
 6                                           , DBMS_RANDOM.VALUE (4, 30)
 7                                                              )
 8                                );
 9     END LOOP;
10* END;
```

Результат будет таким:

```
Random String=RniQZGquFVJYFpGLOvtNd
Random String=GhcphpcsaCXlhigRQY
Random String=JtakoelUf
Random String=BgCOu
Random String=QFBzQxcHqGlHWkZFmnN
Random String=lSxVjqJvpwBB
Random String=jfhNARzALrLOKZRpOwnhrzz
Random String=KuFtdJcqQpjkrFmzFbzcXnYFGjWo
Random String=BhuZ
Random String=GebcqcgvzBfEpTYnJPmYAQdb
```

Теперь строки не только являются случайными, но и имеют случайные разные длины.

Кроме того, необходимо, чтобы 25% фамилий приходились на долю «Smith», а остальные были случайными. Комбинируем случайные строки и метод Монте-Карло:

```
DECODE (
      FLOOR(DBMS_RANDOM.value(1,5)),
         1,'Smith',
         DBMS_RANDOM.string ('A',DBMS_RANDOM.value(4,30))
   )
```

Это выражение в 25 процентах случаев будет возвращать «Smith», а в остальное время – произвольную буквенную строку длиной от 4 до 30 символов.

Сводим воедино

Теперь, когда вы познакомились с составляющими процесса генерирования случайных чисел, сведем их все воедино. Напишем фрагмент PL/SQL-кода формирования записей о счетах. В этом разделе будем загружать 100 000 записей в таблицу ACCOUNTS (приводится полный текст загружающей программы).

```
/* Файл на веб-сайте: ins_acc.sql */
BEGIN
   FOR l_acc_no IN 1 .. 100000
   LOOP
      INSERT INTO accounts
           VALUES (l_acc_no,

                   -- Имя
                   DECODE (FLOOR (DBMS_RANDOM.VALUE (1, 21)),
                           1, 'Alan',
                           2, 'Alan',
                           3, 'Barbara',
                           4, 'Barbara',
                           5, 'Charles',
                           6, 'David',
```

```
        7, 'Ellen',
        8, 'Ellen',
        9, 'Ellen',
        10, 'Frank',
        11, 'Frank',
        12, 'Frank',
        13, 'George',
        14, 'George',
        15, 'George',
        16, 'Hillary',
        17, 'Iris',
        18, 'Iris',
        19, 'Josh',
        20, 'Josh',
        'XXX'
       ),

-- Фамилия
DECODE (FLOOR (DBMS_RANDOM.VALUE (1, 5)),
        1, 'Smith',
        DBMS_RANDOM.STRING ('A'
                            , DBMS_RANDOM.VALUE (4, 30))
       ),

-- Тип счета
DECODE (FLOOR (DBMS_RANDOM.VALUE (1, 5)),
        1, 'S',
        2, 'C',
        3, 'M',
        4, 'D',
        'X'
       ),

-- Идентификатор листа
CASE
   WHEN DBMS_RANDOM.VALUE (1, 100) < 51
      THEN NULL
   ELSE l_acc_no + FLOOR (DBMS_RANDOM.VALUE (1, 100))
END,

-- Тип субсчета
CASE
   WHEN DBMS_RANDOM.VALUE (1, 100) < 76
      THEN NULL
   ELSE DECODE (FLOOR (DBMS_RANDOM.VALUE (1, 6)),
                1, 'S',
                2, 'C',
                3, 'C',
                4, 'C',
                5, 'C',
                NULL
               )
END,
```

```
                    -- Дата открытия счета
                    SYSDATE - DBMS_RANDOM.VALUE (1, 500),
                    -- Дата изменения счета
                    SYSDATE,

                    -- Идентификатор администратора
                    DECODE (FLOOR (DBMS_RANDOM.VALUE (1, 11)),
                            1, 1,
                            2, 1,
                            3, 1,
                            4, 1,
                            5, 2,
                            6, 3,
                            7, 4,
                            8, 5,
                            9, 5,
                            10, 5,
                            0
                           ));
    END LOOP;

    COMMIT;
END;
```

Как узнать, что наши старания привели к нужному результату? Как говорится, не попробуешь – не узнаешь. После загрузки данных проверяем реальное распределение:

```
SQL> SELECT   first_name, COUNT (*)
  2      FROM accounts
  3  GROUP BY first_name
  4  ORDER BY first_name
  5  /

FIRST_NAME                     COUNT(*)
------------------------------ ----------
Alan                             9766
Barbara                         10190
Charles                          5066
David                            5000
Ellen                           15023
Frank                           15109
George                          14913
Hillary                          5019
Iris                             9932
Josh                             9982

10 rows selected.
```

Имена полностью отвечают запрошенному распределению. Например, если помните, мы хотели, чтобы 10% строк содержали имя «Alan». Имеем 9766 таких записей из 100000, что составляет приблизительно 10%. Мы хотели 15% записей с именем «Ellen», а получили 15,023% –

очень близко к запрошенному числу и статистически показательно. Можно проверить все остальные столбцы и убедиться в том, что они действительно распределены по выбранному шаблону.

Заключение

Возможность генерирования случайных значений чрезвычайно важна для множества приложений. Случайный выбор является неотъемлемой частью шифрования, а также играет важную роль в различных других областях. В этой главе показано, как генерировать разнообразные случайные числа и строки различной точности и длины; как обеспечить случайность, используя должные методы задания начального значения последовательности случайных чисел; и как избежать генерирования неслучайных чисел, получаемых при использовании неизменного начального значения. Для того чтобы генерируемые значения лучше отражали картину реального мира, предложены способы создания случайных значений, соответствующих определенным статистическим шаблонам. Наконец, было рассказано о том, как проверить случайность полученных данных.

8

Использование планировщика

Администрирование базы данных требует выполнения множества задач. Некоторые из них запускаются сразу, без подготовки, другие должны выполняться регулярно в предопределенное время. В качестве примера можно привести сбор статистики для объектов базы данных, сбор сведений о свободном пространстве внутри базы данных, анализ проблем с экземпляром базы данных и передача по требованию соответствующей информации напрямую администратору базы данных и т. д. Эти задачи могут относиться как к базе данных, так и к операционной системе (например, проверка свободного пространства файловой системы, проверка доступности других серверов и т. д.). Любой администратор базы данных может привести сотню примеров, когда для некоторой задачи (обычно называемой *заданием* – *job*) необходимо обеспечить выполнение в определенный момент в будущем, а не выполнять ее незамедлительно.

В этой главе будут описаны возможности PL/SQL по управлению планировщиком заданий в базе данных, а также вне базы данных (но на том же сервере). Данная функциональность обеспечивается двумя встроенными пакетами: `DBMS_JOB` и `DBMS_SCHEDULER`. `DBMS_JOB`, появившийся в Oracle8*i*, имеет весьма ограниченные возможности. В Oracle 10*g* Release 1 он фактически заменен пакетом `DBMS_SCHEDULER`. Новый пакет является гораздо более мощным и надежным средством, так что следует использовать его вместо `DBMS_JOB`. Эта глава будет посвящена исключительно использованию `DBMS_SCHEDULER`. Программа Oracle Enterprise Manager с графическим пользовательским интерфейсом, входящая в Oracle 10*g*, позволяет управлять планировщиком заданий, но ее описание лежит за рамками нашего обсуждения.

Планировщик Oracle включает в себя четыре основных компонента, которые схематично представлены на рис. 8.1.

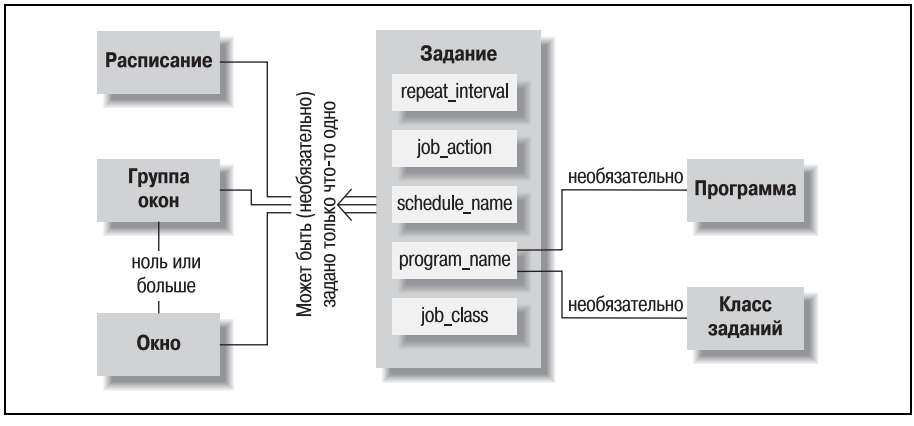


Рис. 8.1. *Составляющие планировщика Oracle*

- Задание – объект, который определяет, какая программа в какое время выполняется.
- Расписание – именованный объект, который хранит календарь с конкретными временами запуска.
- Программа – именованный объект, который определяет, какой исполняемый файл должен выполняться.
- Окно, которое задает определенную длительность и схему выделения ресурсов для задания.

Заданию сопоставляются следующие параметры:

- Параметр `job_action`, который определяет хранимую процедуру, анонимный блок PL/SQL или исполняемый файл операционной системы.
- Параметр `repeat_interval`, который указывает, когда и как часто должно выполняться задание.

Дополнительно можно сопоставить заданию именованное расписание, которое будет содержать календарные строки. Параметр `schedule_name` может задавать расписание, окно или группу окон. Задание может выполнять не только действие, указанное в параметре `job_action`, но и именованную программу, которая ссылается на действия тех же типов, что и параметр `job_action` (это могут быть хранимые процедуры, анонимные блоки PL/SQL или исполняемые файлы операционной системы).

Задание может входить в определенный *класс заданий*. Классы заданий объединяют похожие задания и определяют для них некоторые общие свойства (например, уровень журналирования).

Окно определяет начальный и конечный моменты времени, а также план управления ресурсами. Если заданию сопоставлено окно, то выполнение задания начинается и заканчивается в моменты времени, определяемые окном. При этом выполнение задания регулируется

планом ресурсов, определенным для окна. Окно может быть элементом некоторого *класса окон*. Если заданию сопоставляется класс окон, то на задание действуют все окна класса.

Далее в главе все компоненты планировщика будут рассмотрены подробно и с примерами их использования.

Зачем использовать планировщик заданий Oracle?

Администратор базы данных Oracle, которому необходимо спланировать выполнение задания на какое-то определенное время в будущем или по расписанию через определенные промежутки времени, имеет две возможности для решения задачи:

Использовать планировщик операционной системы

Выполнить программу или сегмент кода из утилиты операционной системы, такой как `cron` в Unix или `AT` в Microsoft Windows. Эти средства позволяют выполнить команду определенной операционной системы. Можно даже запустить задание Oracle из внешнего интерфейса (обычно для этого необходимо создать его как сценарий SQL и запустить в командной строке SQL*Plus).

Использовать планировщик базы данных внутри Oracle

Определить операцию как задание в базе данных, которое должно быть выполнено единожды или периодически повторяться, с помощью пакета `DBMS_SCHEDULER`. Такое задание базы данных выполняется в контексте базы данных, а не операционной системы.

Если у вас есть возможность использовать стандартный планировщик операционной системы, зачем вообще задумываться о планировщике базы данных? Есть ряд весьма убедительных аргументов.

Зависимость от базы данных

Задание операционной системы работает вне зависимости от доступности базы данных. Что же произойдет, если задание по сбору статистики оптимизатора запустится в запланированное время, а база данных в этот момент по какой-то причине будет остановлена? Задание не выполнится, что может привести к ряду неприятных результатов. В случае со сбором статистики мы еще легко отделаемся, в других ситуациях последствия могут быть более серьезными. Предположим, например, что у вас есть задание операционной системы, которое выполняет ряд задач базы данных и операционной системы и в случае неудачи оповещает администратора базы данных. Если база данных остановлена, оповещение все равно выполняется, хотя и некорректно. Конечно, можно добавить в начало кода несколько строк, чтобы проверять доступность базы данных прежде, чем приступать к выполнению, но тогда придется добавлять

этот код в каждую утилиту, что может усложнить сопровождение. А вот задание базы данных будет выполняться только при запущенной базе данных, и никогда – при остановленной. Это обеспечивает более полный и точный контроль над процессом выполнения.

Защита

Задания являются частью базы данных, поэтому обеспечены такой же защитой, как и все другие объекты базы данных. Вы можете выдавать и отзывать привилегии на определенные задания для отдельных пользователей. При резервном копировании базы данных сохраняются и задания с соответствующей метаинформацией (такой как комментарии, расписания и т. д.).

Независимость от операционной системы

Посмотрим на синтаксис команды `cron` в Unix и команды `AT` (Планировщик заданий) в Windows. Они абсолютно разные. Что если придется переносить базу данных с одной платформы на другую? Для соответствия новой платформе придется изменить каждый сценарий, а это весьма трудоемкая работа. Если же осуществлять управление заданиями из базы данных, то определения заданий будут перенесены на новую платформу вместе с базой данных; так что менять что бы то ни было в определении заданий не придется.

Независимость от места хранения утилит

Предположим, что вы сохранили все свои утилиты в каталоге `/u01/app/dbatools`, а затем решили переместить их в другой каталог, `u02/app/dbatools`. Если в качестве планировщика используется `cron` или `AT`, то такое простое изменение потребует внесения изменений в каждое определение вашего расписания. При использовании планировщика базы данных вообще нет необходимости в предоставлении информации о том, где находятся утилиты.

Безопасность

При вызове кода PL/SQL или даже простого сценария SQL из операционной системы необходимо передавать в выполняемое задание имя пользователя и пароль. Давайте рассмотрим простейший случай. Предположим, что у вас есть сценарий оболочки, который вызывает SQL-сценарий `myjob.sql`:

```
$ORACLE_HOME/bin/sqlplus -s scott/tiger @/u01/app/dbatools/myjob.sql
```

В команду открытым текстом передается имя пользователя (`scott`) и его пароль (`tiger`). Любой пользователь может увидеть эту важнейшую информацию при помощи команды `ps -aef`, что, естественно, создает брешь в системе безопасности.

Можно попытаться минимизировать риск раскрытия пароля пользователя Scott, немного изменив сценарий. Поместим имя пользователя и пароль внутрь SQL-сценария `myjob.sql` в качестве информации для подключения к базу данных:

```
CONNECT scott/tiger
blah ... blah ... blah ...
```

Тогда вызывающий сценарий будет выглядеть следующим образом:

```
$ORACLE_HOME/bin/sqlplus -s /nolog @/u01/app/dbatools/myjob.sql
```

Переключатель `/nolog` указывает SQL*Plus, что не следует ожидать ввода регистрационных данных в командной строке. Этот подход чуть лучше: имя пользователя и его пароль не видны при просмотре процесса. Однако если пользователь обратится к файлу сценария, то получит доступ к информации (а вы наверняка не хотите, чтобы пользователи видели чужие пароли).

Последним вариантом может стать использование так называемой OPS$-регистрации, которая позволяет операционной системе проводить аутентификацию пользователей. Может существовать Unix-пользователь `scott` и идентифицируемый извне Oracle-пользователь `OPS$SCOTT`. Задание `cron` выглядело бы следующим образом:

```
$ORACLE_HOME/bin/sqlplus -s / @/u01/app/dbatools/myjob.sql
```

Однако в этом главный недостаток метода. Он не только основан на том, что ваше регистрационное имя в Oracle совпадает с регистрационным именем в операционной системе, но и позволяет тому, кто получает доступ к вашему паролю операционной системы, подключиться к вашей схеме базы данных без аутентификации.

Если используется планировщик базы данных, то подобные сценарии не понадобятся, и пользовательский пароль никогда не будет раскрыт. Многие компании с высокими требованиями к безопасности имеют строгие правила относительно нераскрытия паролей ни при каких условиях. Для них единственным выходом является планирование в базе данных.

Гибкость

Если вы используете планировщик операционной системы и хотите изменить пароль пользователя, то вам придется изменить его во всех необходимых сценариях оболочки. Некоторые компании для минимизации подобных усилий создают общий центральный файл паролей и считывают его для каждого сценария оболочки. Это несколько упрощает работу, но создает другую проблему. Если пользователь хочет изменить пароль, необходимо изменить файл паролей. Если операционная система разрешает доступ к серверу базы данных только для администратора базы данных (а именно так и должно быть), то обычный пользователь не сможет изменить файл паролей и вынужден будет просить администратора о помощи. Привлечение посторонних к управлению своим паролем является весьма нежелательным.

Если используется планировщик базы данных, то в хранении пароля нет необходимости.

В общем и целом, при необходимости выполнения каких-то связанных с Oracle операций проще, надежнее и эффективнее использовать пакет DBMS_SCHEDULER.

Здесь и далее в главе для ссылки на систему планирования Oracle будем использовать названия DBMS_SCHEDULER или Планировщик (Scheduler).

Управление заданиями

Базовой единицей работы в планировании является задание. В этом разделе будет описано создание заданий и управление ими в базе данных при помощи процедуры CREATE_JOB пакета DBMS_SCHEDULER. Начнем с простого примера, иллюстрирующего саму идею планирования выполнения заданий.

Простой пример

Предположим, что мы работаем с банковским приложением и хотим обеспечить выполнение задания, которое ежедневно начисляет проценты на остаток на счете. Для начисления процентов используется хранимая процедура apply_interest. Используем программу CREATE_JOB пакета DBMS_SCHEDULER для создания задания планировщика apply_daily_interest:

```
1  BEGIN
2     DBMS_SCHEDULER.create_job (job_name => 'apply_daily_interest',
3                    job_type             => 'STORED_PROCEDURE',
4                    job_action           => 'apply_interest',
5                    repeat_interval      => 'FREQ=DAILY; INTERVAL=1',
6                    enabled              => TRUE,
7                    comments             => 'Apply Daily Interest'
8                   );
9* END;
```

Давайте подробно рассмотрим строки этого фрагмента:

Строка	Описание
2	Определяем имя задания. Имя должно быть действительным идентификатором Oracle (то есть начинаться с буквы, содержать некоторую комбинацию букв, цифр, символов #, _ или $ и быть длиной не более 30 символов). Имя также должно быть уникальным в рамках пространства имен Oracle (например, имя не может совпадать с названием таблицы).
3	Указываем, программа какого типа будет выполняться. В этом замечательное преимущество пакета DBMS_SCHEDULER по отношению к функциональности DBMS_JOB. Исполняемая программа apply_interest является хранимой процедурой, на что указывает значение STORED_PROCEDURE параметра job_type. Для данного аргумента также разрешены значения PLSQL_BLOCK и EXECUTABLE.

Строка	Описание
4	Параметр job_action показывает, что делает данное задание. В нашем случае задание выполняет хранимую процедуру apply_interest.
5	Параметр repeat_interval указывает, как часто задание должно запускаться. В данном случае ежедневно (об использовании интервалов мы подробно поговорим далее в главе).
6	Значение TRUE (также принимаемое по умолчанию) параметра активации означает, что задание будет незамедлительно активировано и запущено.
7	В этой строке можно указать необязательные комментарии для задания, которые впоследствии помогут понять, что делает задание.

После того как задание создано, сведения о нем можно найти в представлении словаря данных DBA_SCHEDULER_JOBS. Давайте взглянем на это представление, чтобы проверить результат вызова CREATE_JOB (далее в главе представление DBA_SCHEDULER_JOBS будет описано подробно).

```
SELECT owner, job_name, job_type
     , job_action, repeat_interval, comments
  FROM dba_scheduler_jobs
```

Для удобства просмотра приведем вывод в вертикальном формате: в левом столбце представлены названия столбцов, а в правом – их содержимое.

```
OWNER                          : ARUP
JOB_NAME                       : APPLY_DAILY_INTEREST
JOB_TYPE                       : STORED_PROCEDURE
JOB_ACTION                     : apply_interest
REPEAT_INTERVAL                : FREQ=DAILY; INTERVAL=1
COMMENTS                       : Apply Daily Interest
```

Создано задание, которое вызывает хранимую процедуру. Если не принимать во внимание синтаксические отличия, то основная функциональность на первый взгляд кажется абсолютно идентичной предоставляемой пакетом DBMS_JOB. Однако, прочитав последующие разделы, вы поймете, насколько более широкие возможности поддерживает DBMS_SCHEDULER.

Выполнение исполняемых файлов операционной системы и анонимных блоков

Предположим, необходимо создать задание, которое запускает сценарий Unix, а не программу PL/SQL. Типичным примером является инкрементальное резервное копирование RMAN. Полный путь сценария RMAN таков: u01/app/oracle/admin/tools/kick_rman_inc.sh. Будучи исполняемым файлом Unix, сценарий лишь частично попадает под юрисдикцию базы данных Oracle. Необходимо ли обращаться для его исполнения к заданию cron? Как уже говорилось выше, недостаток планировщика операционной системы заключается в том, что задание

будет выполняться даже при остановленной базе данных. В нашем же случае хотелось бы, чтобы задание выполнялось только при запущенной базе данных. Неоспоримым преимуществом пакета DBMS_SCHEDULER по сравнению с DBMS_JOB является его способность напрямую определять и выполнять внешние программы из контекста базы данных.

Рассмотрим пример использования процедуры CREATE_JOB пакета DBMS_SCHEDULER для планирования инкрементального резервного копирования RMAN.

```
 1  BEGIN
 2     DBMS_SCHEDULER.create_job
 3       (job_name        => 'RMAN_INC',
 4       repeat_interval  => 'FREQ=DAILY; BYHOUR=2',
 5       job_type         => 'EXECUTABLE',
 6       job_action       => '/u01/app/oracle/admin/tools/kick_rman_inc.sh',
 7       enabled          => TRUE,
 8       comments         => 'Take RMAN Inc Backup'
 9        );
10* END;
```

Это задание вызывает исполняемый файл UNIX kick_rman_inc.sh. Обратите внимание, что параметр job_type в строке 5 установлен в значение EXECUTABLE.[1]

Процедуру CREATE_JOB можно использовать для создания заданий, вызывающих анонимные блоки кода PL/SQL, например:

```
BEGIN
   DBMS_SCHEDULER.create_job
           (job_name             => 'RMAN_INC',
            repeat_interval      => 'FREQ=DAILY; BYHOUR=2',
            job_type             => 'PLSQL_BLOCK',
            job_action           => 'DECLARE i number; BEGIN code; END;',
            enabled              => TRUE,
            comments             => 'Take RMAN Inc Backup'
            );
END;
```

В данном случае параметр job_action содержит весь PL/SQL-блок DECLARE-BEGIN-END, завершенный точкой с запятой. Вы можете поместить любой корректный анонимный PL/SQL-блок вместо слова *code*.

Представление DBA_SCHEDULER_JOBS

После того как задание создано, информация о нем попадает в представление словаря данных DBA_SCHEDULER_JOBS. Приведем описание некоторых основных столбцов (полный перечень можно найти в приложении А).

OWNER

Владелец задания.

[1] В первом примере параметр был установлен в значение STORED_PROCEDURE.

JOB_NAME

Имя задания.

CLIENT_ID

Если при создании задания пользователь указал идентификатор клиента для сеанса, он будет записан в данный столбец. Для задания идентификатора клиента можно вызвать функцию DBMS_SESSION.SET_IDENTIFIER.

GLOBAL_UID

Если пользователь является глобальным (или корпоративным), то в данный столбец записывается идентификатор глобального пользователя.

JOB_TYPE

Тип задания; допустимы следующие значения: EXECUTABLE, PLSQL_BLOCK и STORED_PROCEDURE.

JOB_ACTION

Что делает задание. Для PL/SQL-блока отображается весь код. Для исполняемого файла и хранимой процедуры отображается название.

START_DATE

Время начала выполнения задания как значение типа TIMESTAMP.

REPEAT_INTERVAL

Календарная строка, задающая расписание выполнения задания (например, FREQ=DAILY; BYHOUR=2). (См. также далее раздел «Календарные строки».)

ENABLED

Активно ли задание (TRUE или FALSE).

STATE

Текущее состояние задания (например, SCHEDULED, RUNNING, SUCCEEDED, FAILED).

RUN_COUNT

Количество запусков задания.

FAILURE_COUNT

Количество неудачных выполнений задания.

RETRY_COUNT

В случае неудачного выполнения задания предпринимается повторная попытка выполнения. В данном столбце отображается количество повторных попыток.

LAST_START_DATE

Временная метка последнего запуска задания.

LAST_RUN_DURATION

Продолжительность последнего выполнения задания.

NEXT_RUN_DATE

Следующая (в расписании) дата выполнения задания.

SYSTEM

Является ли задание системным (TRUE или FALSE).

COMMENTS

Внесенные ранее комментарии.

Дополнительные столбцы будут рассматриваться по мере рассмотрения других аспектов функциональности пакета DBMS_SCHEDULER.

Основы управления заданиями

Мы знаем, как создать задание, теперь поговорим о том, как управлять этим созданным заданием (если не указано иное, то все задачи выполняются различными программами пакета DBMS_SCHEDULER).

Включение и отключение заданий

Временно отключить или включить задание можно при помощи процедур DISABLE и ENABLE соответственно. Например, для того чтобы отключить задание RMAN_INC, поступите следующим образом:

```
BEGIN
    DBMS_SCHEDULER.disable (NAME => 'RMAN_INC');
END;
```

Если это задание в настоящий момент работает, команда возвратит ошибку (по умолчанию). Можно изменить такое поведение, задав параметр FORCE:

```
BEGIN
    DBMS_SCHEDULER.disable (NAME => 'RMAN_INC', FORCE=> TRUE);
END;
```

Присутствие FORCE означает, что задание будет отключено после завершения текущего работающего процесса.

Для того чтобы включить задание, отключенное на текущий момент времени, вызовите процедуру ENABLE.

```
BEGIN
   DBMS_SCHEDULER.enable (NAME => 'RMAN_INC');
END;
```

Если при создании задания не указать параметр ENABLED, то задание будет создано и отключено. Необходимо явно включить задание.

Остановка работающих заданий

Если задание выполняется, и его необходимо остановить, вызовите процедуру STOP_JOB.

```
BEGIN
   DBMS_SCHEDULER.stop_job (JOB_NAME => 'RMAN_INC');
END;
```

Процедура STOP_JOB попытается аккуратно остановить задание. К сожалению, бывают ситуации, в которых «мягкое» завершение невозможно, тогда приведенный выше оператор вернет ошибку. Можно настоять на завершении работы, используя параметр FORCE. Например, для остановки созданного нами ранее задания APPLY_INTEREST (исполняющего хранимую процедуру) можно выполнить следующий блок (задание будет остановлено незамедлительно, даже если оно выполняется в данный момент):

```
BEGIN
   DBMS_SCHEDULER.stop_job (job_name      => 'APPLY_INTEREST',
                            FORCE         => TRUE);
END;
```

Владелец задания может завершить его обычным путем, не задавая параметр FORCE => TRUE. Любой другой пользователь, обладающий привилегиями ALTER для этого задания, также может остановить его обычным путем. Например, владелец задания (ACC_MASTER) может выдать привилегии пользователю ARUP:

```
GRANT ALTER ON apply_daily_interest TO arup;
```

Теперь ARUP также сможет останавливать задание.

Для использования параметра FORCE в вызове функции DBMS_SCHEDULER.STOP_JOB необходимо обладать системной привилегией MANAGE SCHEDULER.

Только задания, выполняющие PL/SQL-блоки и хранимые процедуры, могут быть остановлены с помощью параметра FORCE. Задания, выполняющие исполняемые файлы операционной системы, не могут быть остановлены при помощи параметра FORCE; придется дождаться их завершения.

Запуск задания

Планировщик позволяет явно потребовать запуска задания. Запуск задания вручную может потребоваться в ряде случаев:

- Вы остановили задание, чтобы с чем-то разобраться, а теперь хотите, чтобы оно было завершено.
- Время выполнения задания по расписанию еще не подошло, но вам необходимо выполнить его прямо сейчас.
- Задание выполнилось по расписанию, но оно завершилось с ошибкой, и вы хотите понять ее причины (другими словами, задание запускается для его отладки.)

Запустить задание можно при помощи процедуры RUN_JOB. Например, запустим задание CALCULATE_DAILY_INTEREST:

```
BEGIN
```

```
    DBMS_SCHEDULER.run_job (job_name => 'CALCULATE_DAILY_INTEREST');
END;
```

Задание будет запущено в том сеансе, к которому вы подключены в данный текущий момент, если выполнены следующие условия:

- Подключение осуществлено от имени владельца задания или
- Пользователь имеет системную привилегию ALTER ANY JOB

Процедура RUN_JOB очень удобна. Если задание по какой-то причине не удается выполнить, вы узнаете об этом сразу, не прибегая к исследованию журнала выполнения задания (о журнале мы поговорим в разделе «Управление журналированием»). Даже если задание в текущий момент выполняется по расписанию или если другой пользователь также запускает это задание, процедура RUN_JOB *все равно* позволяет вам незамедлительно запустить задание в вашем сеансе. Запустив задание таким способом, вы сможете выявить любые проблемы, связанные с его выполнением или планированием, так как в случае неудачи соответствующая информация будет выведена непосредственно в среде выполнения.

При вызове DBMS_SCHEDULER.RUN_JOB столбцы RUN_COUNT, LAST_START_DATE, LAST_RUN_DURATION и FAILURE_COUNT представления словаря данных DBA_SCHEDULER_JOBS не обновляются. Другими словами, записи об этом выполнении задания в метаданных не будет.

Предположим, что вы хотите запустить задание, но для его завершения понадобится пять-шесть часов. Вероятно, такое задание лучше запустить как фоновое, вне текущего сеанса. Для этого следует установить параметр use_current_session процедуры RUN_JOB в значение FALSE. Запустим CALCULATE_DAILY_INTEREST как фоновую задачу, затем управление сразу же будет возвращено в сеанс.

```
BEGIN
   DBMS_SCHEDULER.run_job (
       job_name              => 'CALCULATE_DAILY_INTEREST',
        use_current_session   => FALSE
        );
END;
```

Если программа выполняется успешно, то обновляются столбцы RUN_COUNT, LAST_START_DATE, LAST_RUN_DURATION и FAILURE_COUNT представления словаря данных DBA_SCHEDULER_JOBS. Однако если по какой-то причине задание выполнить не удается, информация об этом не появится на экране, а будет записана в журнал выполнения заданий (см. далее раздел «Управление журналированием»).

Удаление задания

Если в задании больше нет необходимости, его можно удалить при помощи процедуры DROP_JOB. Удалим, например, задание APPLY_DAILY_INTEREST:

```
BEGIN
   DBMS_SCHEDULER.drop_job (job_name => 'APPLY_DAILY_INTEREST');
END;
```

Если задание в текущий момент выполняется, то будет возвращена ошибка. Если же задать параметр FORCE, то планировщик сначала попытается остановить задание, а затем удалить его:

```
BEGIN
   DBMS_SCHEDULER.drop_job (job_name => 'APPLY_DAILY_INTEREST',
                            FORCE    => TRUE);
END;
```

Управление календарем и расписанием

Предыдущий раздел был посвящен первому из четырех основных компонентов планировщика: заданию. В этом разделе изучим второй ключевой (и наиболее заметный) компонент – *расписание*, определяющее моменты времени, когда предполагается запускать задание. Расписание можно указать при создании задания в процедуре CREATE_JOB, задав календарную строку в качестве значения параметра repeat_interval или сославшись на уже созданное именованное расписание.

Календарные строки

Синтаксис использования календарной строки для значения параметра repeat_interval процедуры CREATE_JOB чрезвычайно прост, особенно для знающих английский язык. Например, если вы определяете задание, которое выполняется с понедельника по пятницу ровно в 7 часов утра и 3 часа дня, то значением параметра repeat_interval может быть такая календарная строка:

```
FREQ=DAILY; BYDAY=MON,TUE,WED,THU,FRI; BYHOUR=7,15
```

Календарная строка включает в себя два разных по типу предложения: FREQ и BY, разделенные точкой с запятой. Предложение FREQ может быть ровно одно, а предложений BY может быть несколько разных видов, в зависимости от конкретного расписания. Каждое предложение состоит из ключевого слова и значения, которые разделены знаком равенства. Все вместе эти предложения указывают, как часто планируется выполнять задание.

*Частота (*FREQ*)*

Предложение FREQ задает интервал между повторными выполнениями. В нашем случае задание должно повторяться каждый день, поэтому FREQ = DAILY. Допустимы следующие ключевые слова: YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY и SECONDLY.

*Сроки (*BY*...)*

Предложение BY определяет, когда именно задание будет выполняться. В примере, определяя частоту выполнения, мы указали, что зада-

ние выполняется ежедневно (FREQ=DAILY). Теперь следует ограничить множество дней (задание должно выполняться только в рабочие дни), указав BYDAY=MON, TUE, WED, THU, FRI. Далее необходимо потребовать, чтобы задание запускалось в 7 часов утра и в 3 часа дня, указав BYHOUR=7, 15. Разрешены следующие ключевые слова: BYMONTH, BYMONTHDAY, BYYEARDAY, BYHOUR, BYMINUTE и BYSECOND (см. описание в табл. 8.1).

Предположим, что задание следует запускать ежедневно (вне зависимости от дня недели) в 10 часов утра, в 2 часа дня и в 8 часов вечера. Тогда параметр repeat_interval может быть таким:

```
FREQ=DAILY; BYHOUR=10,14,20
```

Что делать, если задание необходимо выполнять *через день*, а не каждый день? Используем новое ключевое слово – INTERVAL:

```
FREQ=DAILY; INTERVAL=2, BYHOUR=10,14,20
```

Предложение INTERVAL=2 изменяет частоту вдвое по отношению к значению, указанному в предложении FREQ. В нашем случае FREQ=DAILY, поэтому задание будет выполняться каждые два дня.

Отрицательные значения означают отсчет от конца периода. Например, следующая строка задает второй час, начиная с конца дня:

```
FREQ=DAILY; BYHOUR=-2
```

Некоторые основные интервальные команды для параметра repeat_interval собраны в табл. 8.1.

Таблица 8.1. Ключевые слова BY для календарных строк в параметре repeat_interval

Ключевое слово	Описание

BYMONTH	Планирование выполнение задания в определенные месяцы. Например, для того чтобы задание вычисления процентов выполнялось только в июне и декабре, задаем:
	`BYMONTH=JUN,DEC`
	Можно использовать не только названия, но и номера месяцев:
	`BYMONTH=6,12`
	Точная дата внутри месяца, когда задание будет выполнено, совпадает с днем его запуска. Например, если задание запускается 4 июля, то в декабре и в июне оно будет выполнено 4 числа. Если необходимо, чтобы задание выполнялось в другой день, следует использовать ключевое слово BYMONTHDAY.

BYMONTHDAY	Указывает, в какой именно день месяца должно выполняться задание. Например, для выполнения задания первого числа каждого месяца в 3 часа дня указываем: `FREQ=MONTHLY; BYMONTHDAY=1; BYHOUR=15` Если опустить предложение BYHOUR, задание по умолчанию будет выполняться в полночь.

Ключевое слово	Описание
BYYEARDAY	Указывает, в какой день года должно выполняться задание. Например, для выполнения задания в пятнадцатый день каждого года задаем: `FREQ=YEARLY; BYYEARDAY=15`

BYHOUR	Планирование выполнения задания в определенные часы. Например, для каждодневного выполнения задания в 3, 6 и 9 часов утра задаем:
	`FREQ=DAILY; BYHOUR=3,6,9`
	Расписание, основанное на такой календарной строке, будет выглядеть следующим образом:
	`07/06/2005 03:00:00`
	`07/06/2005 06:00:00`
	`07/06/2005 09:00:00`
	`07/07/2005 03:00:00`
	`07/07/2005 06:00:00`
	`07/07/2005 09:00:00`
	... и так далее.
	Можно изменить предыдущий пример следующим образом:
	`FREQ=MINUTELY; BYHOUR=3,6,9`
	Частота `MINUTELY` означает, что задание будет выполняться каждую минуту. В этом случае расписание будет таким:
	`07/06/2005 03:00:00`
	`07/06/2005 03:01:00`
	`07/06/2005 03:02:00`
	`07/06/2005 03:03:00`
	`07/06/2005 03:04:00`
	`07/06/2005 03:05:00`
	... и так далее до 03:59:00. Последовательность будет повторена в 6:00:00 вечера и будет длиться до 06:59:00 вечера.

BYMINUTE	Планирование выполнения задания в определенную минуту. Для выполнения задания каждые полчаса задаем такую календарную строку:
	`FREQ=MINUTELY; BYMINUTE=30`
	Расписание, основанное на такой календарной строке, будет выглядеть следующим образом:
	`07/06/2005 00:30:00`
	`07/06/2005 01:30:00`
	`07/06/2005 02:30:00`
	... и так далее.
	Если в данном случае написать `FREQ=HOURLY` вместо `MINUTELY`, то ничего не изменится, так как составляющая часа не указывается. Однако если задать такую строку:
	`FREQ=DAILY; BYMINUTE=30`

Ключевое слово	Описание
	округление будет вестись с точностью до дня. Поэтому расписание будет следующим: 07/06/2005 00:30:00 07/07/2005 00:30:00 07/08/2005 00:30:00 07/09/2005 00:30:00 ... и так далее. Здесь задание выполняется каждый день в 00:30, так как в предложении FREQ указано DAILY.
BYSECOND	Аналогично BYMINUTE и BYHOUR, ключевое слово BYSECOND служит для планирования выполнения задания в определенную секунду.

Отрицательное значение какого-то ключевого слова означает отсчет от конца (а не от начала) периода. Например, следующая календарная строка

```
FREQ=YEARLY; BYYEARDAY=-1
```

указывает на первый день года, начиная с его конца. Поэтому такое задание будет выполняться каждый год 31 декабря.

Примеры календарных строк

Давайте рассмотрим еще несколько примеров календарных строк. Предположим, что все они изначально указаны для заданий, запускаемых 5 июля 2005 года. Покажем, как задавать календарные строки, начиная с определения больших интервалов, а затем переходя к более мелким.

`FREQ=YEARLY`

Задано ежегодное выполнение, так что задание будет работать один раз в год. Никакие даты не указаны, так что по умолчанию задание будет выполняться 5 июля каждый год, начиная с 2006. Это задание будет выполняться каждый год 5 июля в полночь.

`FREQ=YEARLY; INTERVAL=2`

Использовано предложение INTERVAL, поэтому формируется расписание для выполнения не каждый год, а через год. Данное задание будет выполнено 5 июля в 2007 году, 2009, 2011 и т. д.

`FREQ=YEARLY; BYMONTH=JAN`

Наличие предложения BYMONTH означает, что ежегодное выполнение задания не будет по умолчанию приходиться на текущий месяц (июль). Но так как конкретная дата не указана, день все-таки будет взят по умолчанию – пятое число. Таким образом, это задание будет выполняться ежегодно 5 января (опять-таки в полночь).

`FREQ=YEARLY; BYMONTH=JAN; BYMONTHDAY=2`

Обратите внимание на дополнительное предложение `BYMONTHDAY`. Оно указывает на то, что расписание будет действовать для второго дня месяца (а не для дня, предусмотренного по умолчанию). Это задание будет выполняться 2 января 2006 года, 2 января 2007 года и т. д.

`FREQ=YEARLY; BYMONTH=JAN; BYDAY=SUN`

Здесь вместо даты указан день недели внутри определенного месяца. Такое задание будет выполняться ежегодно каждое воскресенье января. Приведем несколько дат выполнения в формате мм/дд/гггг:

```
01/01/2006
01/08/2006
01/15/2006
01/22/2006
01/29/2006
01/07/2007
```

Все это воскресенья.

`FREQ=YEARLY; BYMONTH=JAN; BYMONTHDAY=2; BYDAY=SUN`

Скомбинируем два предыдущих примера расписания. Предложение `BYMONTHDAY` задает определенный день месяца (второй), а предложение `BYDAY` – определенный день недели (воскресенье). Поэтому задание будет выполняться каждое воскресенье января, если оно выпадет на 2-е число. Расписание будет таким:

```
01/02/2011
01/02/2022
01/02/2028
01/02/2033
```

Первый раз задание будет выполнено 2 января 2011, так как это воскресенье.

`FREQ=YEARLY; BYYEARDAY=60`

Запрошено выполнение задания каждый 60-й день года. Расписание выглядит следующим образом:

```
03/01/2006
03/01/2007
02/29/2008
03/01/2009
```

Обратите внимание, что планировщик учел високосность 2008 года: в нем 60-й день выпадает на 29 января. В остальных случаях шестидесятым днем года является 1 марта.

`FREQ=YEARLY; BYYEARDAY=60; BYMONTHDAY=1`

Комбинируя условия, можно получить интересующие нас результаты. Пусть задание выполняется в 60-й день года и в первый день месяца. Получаем такое расписание:

```
03/01/2006
```

```
03/01/2007
03/01/2009
03/01/2010
```

Как видите, 2008 год исключен. В этом году 60-й день выпадает на 29 февраля (то есть не на первый день месяца), поэтому год не включается в расписание.

```
FREQ=YEARLY; BYWEEKNO=2
```

Указываем, что задание должно выполняться каждый день второй недели года. Расписание будет таким:

```
01/09/2006
01/10/2006
01/11/2006
01/12/2006
01/13/2006
01/14/2006
01/15/2006
01/08/2007
```

Все семь дней второй недели каждого года присутствуют в расписании.

```
FREQ=DAILY; BYHOUR=3,6,9
```

Это задание выполняется ежедневно ровно в три, шесть и девять часов утра.

```
07/06/2005 03:00:00
07/06/2005 06:00:00
07/06/2005 09:00:00
07/07/2005 03:00:00
… и так далее.
```

```
FREQ=DAILY; INTERVAL=2; BYHOUR=3,6,9
```

К предыдущим условиям добавлено предложение `INTERVAL=2`. Это означает, что интервал между двумя последовательными выполнениями должен быть вдвое больше интервала, указанного в предложении `FREQ` (то есть в нашем случае должен составлять 2 дня).

```
07/06/2005 03:00:00
07/06/2005 06:00:00
07/06/2005 09:00:00
07/08/2005 03:00:00
… и так далее.
```

Задание выполняется 6 июля, затем 8 июля.

```
FREQ=DAILY; BYHOUR=3,6,9; BYMINUTE=00,15,30,45
```

Это задание, как и предыдущее, выполняется ежедневно в 3, 6 и 9 часов дня, но наличие предложения `BYMINUTE` указывает на то, что задание должно выполняться в указанные минуты. Так что расписание будет выглядеть так:

```
07/06/2005 03:00:00
07/06/2005 03:15:00
07/06/2005 03:30:00
… и так далее.
```

Нумерация недель ISO-8601

Если использовано ключевое слово `BYWEEKNO`, то планировщик применяет нумерацию недель, зафиксированную в стандарте ISO-8601 и имеющую ряд специфических особенностей. Вот что вам необходимо знать о нумерации недель согласно ISO-8601:

- Первым днем недели считается не воскресенье, а понедельник, так как именно такой порядок принят во многих компаниях и программах.
- Недели имеют номера от 1 до 53.
- Первая неделя начинается в первый понедельник года. Следовательно, несколько дней могут предшествовать первой неделе года. Например, если в 2005 году ваша календарная строка выглядит как «`FREQ=YEARLY; BYWEEKNO=1`», то это расписание начнет действовать 2 января 2005, так как это и будет первый понедельник 2005 года. Это означает, что 1 января никогда не появится ни в одном из расписаний. Возможно, для вас это и не важно, но следует иметь это в виду.
- Аналогично последняя неделя года может захватывать не все дни. Часть года может остаться вне последней недели: она будет включена в первую неделю следующего года.
- Последняя неделя года может содержать часть следующего года. Рассмотрим календарную строку «`FREQ=YEARLY; BYWEEKNO=52`». Если она задается в 2005 году, то будет составлено такое расписание:

```
12/26/2005
12/27/2005
12/28/2005
12/29/2005
12/30/2005
12/31/2005
01/01/2006
12/25/2006
... и так далее
```

- Обратите внимание, что последняя неделя года включает в себя 1 января 2006 года, которое относится к 2006, а не 2005 году. Хотя неделя и считается 52-й неделей 2005 года, фактически расписание заходит в 2006 год. Если при составлении расписания вы не будете учитывать такую возможность, то результат может оказаться далеким от ожидаемого.

Календарные строки для дат в будущем

Календарные строки так похожи на обычный английский язык, что писать их не составляет труда. Однако можно случайно что-то перепутать, и в результате расписание окажется не таким, как предполагалось. Процедура EVALUATE_CALENDAR_STRING пакета DBMS_SCHEDULER анализирует календарные строки и формирует тестовое расписание, которое можно тщательно изучить и убедиться в том, что задание будет выполняться так, как предполагалось.

Процедура принимает четыре параметра:

calendar_string

Обрабатываемая календарная строка.

start_date

Дата (в виде значения типа TIMESTAMP), с которой следует начать.

return_date_after

Если необходимо указывать даты и моменты времени, стоящие после какой-то определенной даты, то используйте этот параметр для начала последовательности планируемых после данной даты дат.

next_run_date

Это выходной параметр. Процедура записывает в него дату и время, когда планировщик будет выполнять данную календарную строку.

Давайте рассмотрим пример использования этой процедуры для получения расписания выполнений задания для календарной строки «FREQ=MONTHLY;INTERVAL=2».

```
/* File on web: cal_eval.sql */
DECLARE
   l_start_date    TIMESTAMP;
   l_next_date     TIMESTAMP;
   l_return_date   TIMESTAMP;
BEGIN
   l_start_date := TRUNC (SYSTIMESTAMP);
   l_return_date := l_start_date;

   FOR ctr IN 1 .. 10
   LOOP
      dbms_scheduler.evaluate_calendar_string ('FREQ=MONTHLY;INTERVAL=2',
                                               l_start_date,
                                               l_return_date,
                                               l_next_date
                                              );
      DBMS_OUTPUT.put_line (   'Next Run on: '
                || TO_CHAR (l_next_date, 'mm/dd/yyyy hh24:mi:ss')
                );
      l_return_date := l_next_date;
   END LOOP;
END;
```

Результат будет таким:

```
Next Run on: 09/06/2005 00:00:00
Next Run on: 11/06/2005 00:00:00
Next Run on: 01/06/2006 00:00:00
Next Run on: 03/06/2006 00:00:00
Next Run on: 05/06/2006 00:00:00
Next Run on: 07/06/2006 00:00:00
Next Run on: 09/06/2006 00:00:00
Next Run on: 11/06/2006 00:00:00
Next Run on: 01/06/2007 00:00:00
Next Run on: 03/06/2007 00:00:00
```

Просмотрев точные даты и моменты времени будущих выполнений задания, вы сможете проверить, совпадают ли они с ожидаемыми.

Именованные расписания

Календарные строки очень полезны при составлении расписания выполнения ваших заданий. Но предположим, что существует несколько заданий, выполняющихся одновременно (например, сбор статистики оптимизатора для нескольких таблиц). Приведем фрагмент представления словаря данных, отображающий расписание для таких заданий:

```
SQL> SELECT job_name, repeat_interval
  2  FROM dba_scheduler_jobs;

JOB_NAME                       REPEAT_INTERVAL
------------------------------ ------------------------------
TABSTAT_ACCOUNTS               FREQ=DAILY; BYHOUR=3
TABSTAT_SAVINGS                FREQ=DAILY; BYHOUR=3
TABSTAT_CHECKING               FREQ=DAILY; BYHOUR=3
... и так далее ...
```

Как видите, все задания имеют одну и ту же календарную строку – «FREQ=DAILY; BYHOUR=3», которая означает ежедневное выполнение задания в 3 часа дня. Если теперь нужно будет изменить время с 3 часов на 4 – что вы будете делать?

Придется очень аккуратно пройтись по всем календарным строкам и изменить каждую из них. Чем больше заданий, чем больше работы придется проделать; это будет весьма утомительно, и к тому же возможны ошибки. Это общий недостаток любого жесткого кодирования внутри приложения.

Планировщик позволяет избежать жесткого кодирования за счет создания *именованного* расписания, на которое смогут ссылаться все задания. При использовании именованного расписания нет необходимости в явном указании календарной строки: поддержка расписания значительно упрощается. Давайте посмотрим, как это работает.

Используя процедуру CREATE_SCHEDULE, создаем расписание с именем opt_stat_coll_sched, которое задает приведенную ранее календарную строку

```
1  BEGIN
2     DBMS_SCHEDULER.create_schedule
3        (schedule_name       => 'opt_stat_coll_sched',
4        start_date           => SYSTIMESTAMP,
5        repeat_interval      => 'FREQ=DAILY; BYHOUR=3',
6        comments             => 'Run daily at 3 AM'
7         );
8  END;
```

Теперь посмотрим на строки подробно.

Строка	Описание
3	Имя расписания.
4	Время начала действия именованного расписания. В этом примере расписание должно начинать действовать в текущий момент. Время должно задаваться как значение типа TIMESTAMP, поэтому использовано значение SYSTIMESTAMP вместо SYSDATE.
5	Календарная строка, определяющая расписание.
6	Комментарии, описывающие расписание.

После того как расписание создано, сценарий создания задания может на него ссылаться по имени. Давайте удалим уже созданные задания и начнем заново:

```
BEGIN
   DBMS_SCHEDULER.drop_job (job_name => 'tabstat_savings');
END;
```

Удаляем все задания и пересоздаем их с использованием именованного расписания. Вот пример для задания TABSTAT_SAVINGS.

```
1  BEGIN
2     DBMS_SCHEDULER.create_job (job_name  => 'tabstat_savings',
3                job_type           => 'stored_procedure',
4                job_action         => 'collect_stats_checking',
5                schedule_name      => 'opt_stat_coll_sched',
6                enabled            => TRUE,
7                comments           => 'Collect Optimizer Stats'
8                 );
9* END;
```

Код практически идентичен коду предыдущего фрагмента создания задания. Отличие имеется в строке 5, где появляется новый параметр: schedule_name, в котором можно ссылаться на только что созданное расписание.

Для того чтобы проверить сопоставленное заданию расписание, можно (как и раньше) обратиться к представлению словаря данных DBA_SCHEDULER_JOBS. Предположим, что задание TABSTAT_SAVINGS создано заново с использованием именованного расписания. Два других задания все еще сопоставлены календарным строкам.

```
SQL> SELECT job_name, repeat_interval, schedule_name, schedule_owner,
                                                      schedule_type
  2  FROM dba_scheduler_jobs;
                                                       SCHEDULE_
JOB_NAME          REPEAT_INTERVAL     SCHEDULE_NAME      OWNER     SCHEDULE_TYPE
----------------- ------------------- ------------------ --------- ------------
TABSTAT_ACCOUNTS  FREQ=DAILY; BYHOUR=3                              CALENDAR
TABSTAT_SAVINGS                       OPT_STAT_COLL_SCHED  ARUP     NAMED
TABSTAT_CHECKING  FREQ=DAILY; BYHOUR=3                              CALENDAR
```

Как видите, столбец SCHEDULE_TYPE отображает значение NAMED для задания, которому было сопоставлено именованное расписание. Для остальных заданий (продолжающих использовать календарные строки) в этом столбце выводится значение CALENDAR. Столбец SCHEDULE_NAME содержит имя расписания, сопоставленного заданию. Кроме того, при сопоставлении заданию именованного расписания столбец REPEAT_INTERVAL для такого задания устанавливается в значение NULL.

Владелец расписания

Еще один важный столбец представления DBA_SCHEDULER_JOBS — это SCHEDULE_OWNER. В нашем примере он содержит значение ARUP, имя пользователя, который создал расписание. Наличие этого столбца отражает важный факт: расписание не обязательно должно быть создано пользователем, фактически запускающим задание, оно может быть определено другим пользователем. Однако имейте в виду, что пользователь, создающий расписание, должен иметь системную привилегию CREATE JOB. Расписание создается средствами пакета DBMS_SCHEDULER, поэтому пользователь также должен обладать привилегией EXECUTE на пакет.

Предположим, что я создаю специального пользователя SCHED_MANAGER для управления всеми расписаниями базы данных. Такой подход упрощает управление расписаниями и позволяет контролировать их из одной точки. Как уже говорилось, пользователю должны быть выданы привилегии CREATE JOB, EXECUTE ON DBMS_SCHEDULER.

Код создания задания теперь будет выглядеть следующим образом:

```
 1  BEGIN
 2     DMS_SCHEDULER.create_job
 3           (job_name           => 'tabstat_savings',
 4            job_type           => 'stored_procedure',
 5            job_action         => 'collect_stats_checking',
 6            schedule_name      => 'SCHED_MANAGER.opt_stat_coll_sched',
 7            enabled            => TRUE,
 8            comments           => 'Collect SAVINGS Stats'
 9            );
10* END;
```

Обратите внимание на изменение строки 6. Теперь параметр schedule_name содержит значение SCHED_MANAGER.opt_stat_coll_sched, указывающее на владельца расписания. В предыдущих примерах мы не вклю-

чали в название расписания префикс (если префикс не указан, то по умолчанию считается, что расписание принадлежит пользователю, создающему задание).

После того как расписание создано некоторым пользователем, использовать его может *произвольный* другой пользователь. Не существует средств детального контроля доступа, которые обеспечивали бы управление доступом к определенному расписанию. Помните об этом при создании расписаний.

Управление именованными программами

Мы изучили два основных компонента планировщика: задание и расписание. Третьим базовым элементом является *программа*: тот код, который будет выполняться по расписанию. Вы уже научились вызывать программы разных типов, теперь можно попробовать (и это вполне оправданно) поместить все планируемые процессы (в хранимых процедурах, анонимных блоках PL/SQL или исполняемых файлах) под контроль планировщика. Однако тут возникает одна сложность. В соответствии с предложенным ранее описанием синтаксической конструкции вам необходимо передать полный путь к исполняемому файлу (или к PL/SQL-блоку или хранимой процедуре) в параметр `job_action` процедуры `CREATE_JOB`. Если этот путь длинный (а именно так зачастую и бывает), то вводить его полностью не слишком удобно. А если, к тому же, приходится вводить его неоднократно, то увеличивается вероятность ошибок при вызове планировщика.

К счастью, `DBMS_SCHEDULER` позволяет определять *именованные* программы, что значительно упрощает ссылку на выполняемую программу. Именованная программа, по сути, – синоним кода, который фактически выполняется. Вместо того чтобы вызывать реальный сегмент кода, вы ссылаетесь на программу по имени. Теперь, если когда-нибудь нужно будет изменить задание так, чтобы оно запускало другую программу, вам не придется изменять определение задания. Необходимо заменить исполняемый модуль внутри программы, и задание будет работать, как надо.

Создание программы

В следующем примере используем процедуру `CREATE_PROGRAM` пакета `DBMS_SCHEDULER` для получения периодически вызываемой исполняемой программы с именем `KICK_RMAN_INC`.

```
BEGIN
   DBMS_SCHEDULER.create_program
          (program_name    => 'KICK_RMAN_INC',
           program_type    => 'EXECUTABLE',
           program_action  => '/u01/app/oracle/admin/tools/kick_rman_inc.sh',
           enabled         => TRUE,
```

```
            comments        => 'Take RMAN Inc Backup'
           );
END;
```

После создания такой программы в созданном ранее задании на инкрементальное резервное копирование RMAN не придется указывать полный путь программы и тип задания. Новое определение будет таким:

```
BEGIN
   DBMS_SCHEDULER.create_job (job_name           => 'RMAN_INC',
                              program_name       => 'KICK_RMAN_INC',
                              schedule_name      => 'EVERY_DAY',
                              comments           => 'RMAN Inc Backup',
                              enabled            => TRUE
                             );
END;
```

Как видите, указаны только имя расписания и имя программы. Программа (и полный путь к ней) уже определена как исполняемый файл, поэтому не следует указывать, к какому типу программ она относится (PL/SQL-блок, хранимая процедура или исполняемый файл). Соответственно, параметр `program_type` исключен из описания задания. Более того, после того как программа определена, ее также можно использовать в другом задании без необходимости задания каких бы то ни было деталей. Можно вообще полностью заменить программу (например, сделать блоком PL/SQL или хранимой процедурой вместо исполняемого файла операционной системы): никакие изменения в коде создания задания не потребуются.

Запуск программ других пользователей

При обсуждении расписаний уже упоминалось, что программы могут принадлежать пользователям, отличным от их создателей.[1] По умолчанию предполагается, что определяемая в процедуре `CREATE_JOB` программа принадлежит пользователю, создающему задание. Однако при желании можно использовать программу, принадлежащую другому пользователю. Предположим, например, что пользователь `INTEREST_ADMIN` владеет программой начисления процентов:

```
SQL> CREATE USER interest_admin IDENTIFIED BY interest_admin;
User created.

SQL> GRANT CREATE SESSION, CREATE JOB, CREATE PROCEDURE to interest_admin;
Grant succeeded.

SQL> CONN interest_admin/interest_admin
Connected.
```

[1] Автор имеет в виду, что создать хранимую процедуру может один пользователь, именованную программу на ее основе – другой, а задание на основе этой программы – третий. Что и демонстрируется в нижеследующем примере. – *Примеч. науч. ред.*

```
SQL> CREATE PROCEDURE calc_int
  2  AS
  3  BEGIN
  4     NULL;
  5  END;
  6  /

Procedure created.
```

Используем эту процедуру в именованной программе CAL_INTEREST от имени пользователя INTEREST_ADMIN:

```
BEGIN
   DBMS_SCHEDULER.create_program (program_name      => 'CALC_INTEREST',
                                  program_type      => 'STORED_PROCEDURE',
                                  program_action    => 'calc_int',
                                  enabled           => TRUE,
                                  comments          => 'Calculate Interest'
                                 );
END;
```

После того как программа создана, создаем расписание EVERY_DAY, принадлежащее пользователю SCHED_MANAGER.

```
SQL> CONN sched_manager/sched_manager
Connected.
SQL> BEGIN
  2     DBMS_SCHEDULER.create_schedule (schedule_name    => 'every_day',
  3                                     start_date       => SYSTIMESTAMP,
  4                                     repeat_interval  => 'FREQ=DAILY; BYHOUR=3',
  5                                     comments     => 'Schedule Run for Int Calc'
  6                                             );
  7  END;
  8  /

PL/SQL procedure successfully completed.
```

Прежде чем пользователь ACC_MANAGER сможет использовать эту программу, ему должны быть предоставлены привилегии EXECUTE на программу или системная привилегия EXECUTE ANY PROGRAM.

```
SQL> CONN interest_admin/interest_admin
Connected.

SQL> GRANT EXECUTE ON calc_interest TO acc_manager;
Grant succeeded.
```

Теперь создадим задание от имени пользователя ACC_MANAGER.

```
BEGIN
   DBMS_SCHEDULER.create_job (job_name    => 'Calculate_Daily_Interest',
                              program_name   => 'INTEREST_ADMIN.CALC_INTEREST',
                              schedule_name  => 'SCHED_MANAGER.EVERY_DAY',
                              comments       => 'Daily Interest Calculation',
                              enabled        => TRUE
```

```
                                    );
END;
```

Создается задание `Calculate_Daily_Interest`, которое выполняет программу, на которую ссылается именованная программа `CALC_INTEREST`, принадлежащая пользователю `INTEREST_ADMIN`, причем выполняет ее по именованному расписанию `EVERY_DAY`, принадлежащему пользователю `SCHED_ADMIN`. Ничего себе!

Возможность выполнения именованных программ регулируется привилегией `EXECUTE`. Для контроля ссылок на именованные расписания, принадлежащие другим пользователям, привилегий не существует.

Использование именованных программ может значительно снизить административные накладные расходы. Вам не придется менять определение задания при изменении исполняемого файла, пути или имени хранимой процедуры.

Управление приоритетами

Oracle позволяет присваивать заданиям приоритеты. Эта функциональность чрезвычайно важна для создания прочного и надежного механизма планирования.

Предположим, что в базе данных создан ряд заданий с разнообразными расписаниями. Всегда есть вероятность того, что некоторые задания будут перекрываться. В конце концов, в сутках всего 24 часа. Если два задания выполняются одновременно, они могут потреблять процессорное время и память с интенсивностью, не характерной для обычной работы базы данных. Потребление ресурсов (или, если точнее, конкуренция за ресурсы) может привести к тому, что оба задания будут замедляться (как и остальные приложения, работающие в системе). Как следует поступать в подобных ситуациях? Можно выделить заданию A все необходимые ему ресурсы и заставить задание B немного пострадать, пока A не завершится. Или наоборот, выделить больше ресурсов заданию B.

Использование диспетчера ресурсов

Приоритеты заданий планировщика Oracle управляются диспетчером ресурсов (Resource Manager), доступ к которому обеспечивает встроенный пакет базы данных `DBMS_RESOURCE_MANAGER`. Описание диспетчера ресурсов выходит за рамки нашего обсуждения: будем считать, что вы знакомы с его возможностями.

Давайте рассмотрим пример. Сначала определяем группу диспетчера ресурсов `OLTP_GROUP` для управления нагрузкой на базу данных при выполнении OLTP-операций.

```
BEGIN
   DBMS_RESOURCE_MANAGER.clear_pending_area ();
   DBMS_RESOURCE_MANAGER.create_pending_area ();
   DBMS_RESOURCE_MANAGER.create_consumer_group
          (consumer_group      => 'oltp_group',
           COMMENT             => 'OLTP Activity Group'
          );
   DBMS_RESOURCE_MANAGER.submit_pending_area ();
END;
```

Далее необходимо определить директивы по плану распределения ресурсов, которые указывают, какими образом ресурсы распределяются между различными группами внутри плана. Oracle поставляется с предопределенной группой OTHER_GROUP. В нашем случае мы создали вторую группу, OLTP_GROUP, которая будет объединять все OLTP-операции. Выделяем 80% мощности процессора на OLTP-операции, а оставшуюся часть отдаем OTHER_GROUP.

```
BEGIN
   DBMS_RESOURCE_MANAGER.clear_pending_area ();
   DBMS_RESOURCE_MANAGER.create_pending_area ();
   DBMS_RESOURCE_MANAGER.create_plan ('OLTP_PLAN',
                                      'OLTP Database Activity Plan'
                                     );
   DBMS_RESOURCE_MANAGER.create_plan_directive
                        (PLAN                       => 'OLTP_PLAN',
                         group_or_subplan           => 'OLTP_GROUP',
                         COMMENT                 => 'This is the OLTP Plan',
                         cpu_p1                     => 80,
                         parallel_degree_limit_p1   => 4,
                         switch_group               => 'OTHER_GROUPS',
                         switch_time                => 10,
                         switch_estimate            => TRUE,
                         max_est_exec_time          => 10,
                         undo_pool                  => 500
                                                );
   DBMS_RESOURCE_MANAGER.create_plan_directive
                           (PLAN                       => 'OLTP_PLAN',
                            group_or_subplan           => 'OTHER_GROUPS',
                            COMMENT                    => NULL,
                            cpu_p1                     => 20,
                            parallel_degree_limit_p1   => 0,
                            active_sess_pool_p1        => 0,
                            queueing_p1                => 0,
                            switch_estimate            => FALSE,
                            max_est_exec_time          => 0,
                            undo_pool                  => 10
                           );
   DBMS_RESOURCE_MANAGER.submit_pending_area ();
END;
```

После того как ресурсные планы созданы, их можно использовать при определении заданий.

Существуют два разных способа использования ресурсных планов: с помощью классов заданий или с помощью окон планировщика. Следующий раздел будет посвящен классам заданий. Далее в разделе «Управление окнами» будет рассказано об использовании окон с DBMS_SCHEDULER (в том числе и об их использовании с ресурсными планами).

Класс заданий

Класс заданий – это коллекция свойств. Создать класс заданий можно при помощи процедуры CREATE_JOB_CLASS пакета DBMS_SCHEDULER. Потом при создании задания можно указать имя уже созданного класса заданий в параметре job_class процедуры CREATE_JOB. Рассмотрим пример создания класса заданий планировщика, который связан с группой пользовательских ресурсов OLTP_GROUP.

```
1  BEGIN
2     DBMS_SCHEDULER.create_job_class
3             (job_class_name              => 'OLTP_JOBS',
4              logging_level              => dbms_scheduler.logging_full,
5              log_history                => 45,
6              resource_consumer_group    => 'OLTP_GROUP',
7              comments                   => 'OLTP Related Jobs'
8             );
9  END;
```

Рассмотрим эти строки подробно.

Строка	Описание
3	Имя класса заданий.
4	При выполнении задания ведется журнал операций в принадлежащей пользователю SYS таблице SCHEDULER$_EVENT_LOG. Эта таблица доступна другим пользователем через представление DBA_SCHEDULER_JOB_LOG. Записи журнала могут быть полными или частичными (см. далее раздел «Управление журналированием»).
5	Если выполнение задания протоколируется, то журнальные записи должны периодически удаляться, иначе журнал станет неуправляемо большим. Параметр log_history определяет, сколько дней журнальные записи должны храниться. Значение по умолчанию – 30 (дней). В нашем примере это значение увеличено до 45.
6	Класс заданий связан с группой ресурсов OLTP_GROUP, которая управляет выделением ресурсов и использованием заданий, определенных в данном классе заданий.
7	Комментарии для данного класса заданий.

После того как класс заданий создан, вы можете просмотреть его атрибуты (а также атрибуты определенных ранее классов заданий) в пред-

ставлении словаря данных DBA_SCHEDULER_JOB_CLASSES. Приведем запись из этого представления (отформатированную по вертикали для удобочитаемости):

```
JOB_CLASS_NAME              : OLTP_JOBS
RESOURCE_CONSUMER_GROUP     : OLTP_GROUP
SERVICE                     :
LOGGING_LEVEL               : FULL
LOG_HISTORY                 : 45
COMMENTS                    : OLTP Related Jobs
```

При определении класса заданий можно также определить имя сервиса, под которым будут выполняться задания данного класса. Имя сервиса должно быть определено в базе данных. Например, если используется имя сервиса INTCALC_SRVC, то оно должно быть активировано следующим образом:

```
ALTER SYSTEM SET SERVICE_NAME = 'INTCALC_SRVC';
```

После этого можно определить класс заданий под таким именем сервиса, указав параметр service для процедуры CREATE_JOB_CLASS (см. строку, выделенную жирным шрифтом).

```
BEGIN
   DBMS_SCHEDULER.create_job_class
             (job_class_name      => 'INT_JOBS',
              logging_level       => dbms_scheduler.logging_full,
              log_history         => 45,
              service             => 'INTCALC_SRVC',
              comments            => 'Interest Calculation Related Jobs'
              );
END;
```

Имена сервисов удобны, когда необходимо управлять сеансом, связанным с сервисом, а не с экземпляром. Внутри экземпляра Oracle может быть определено несколько сервисов. При открытии сеанса возможно задание параметра SERVICE_NAME в файле TNSNAMES.ORA вместо использования SID в строке соединения. Если соединение устанавливается именно таким способом, то столбец SERVICE_NAME представления V$SESSION отображает имя сервиса для сеанса. Если сервис в экземпляре отключен, сеансы не будут открыты даже при работающем экземпляре.

Сервисы полезны и для других целей. Например, если вы работаете с многоэкземплярной базой данных RAC, то можете захотеть определить сервис только в одном экземпляре базы данных, тем самым ограничив задание только одним конкретным экземпляром. При отключении экземпляра сеансы передаются другим «выжившим» экземплярам. Определение сервиса дает возможность указать, какому узлу следует передать сеансы при проблемах с исходным экземпляром базы данных. Сервис также можно использовать для управления ресурсными планами сеансов, которым сервис сопоставлен.

Так как сервис также управляет выделением ресурсов, нельзя задавать сразу два параметра: resource_consumer_plan и service. Следует выбрать какой-то один подход и задать соответствующий параметр.

Последним этапом будет создание задания, которое определяется как элемент созданного класса заданий. Создаем задание по сбору статистики оптимизатора.

```
BEGIN
   DBMS_SCHEDULER.create_job (job_name         => 'COLLECT_STATS',
                              job_type         => 'STORED_PROCEDURE',
                              job_action       => 'collect_opt_stats',
                              job_class        => 'OLTP_JOBS',
                              repeat_interval  => 'FREQ=WEEKLY; INTERVAL=1',
                              enabled          => TRUE,
                              comments         => 'Collect Optimizer Stats'
                                 );
END;
```

Обратите внимание на выделенную строку: в ней единственное отличие от уже рассмотренных примеров. Параметр job_class установлен в значение OLTP_JOBS (созданный нами класс заданий). Задание COLLECT_STATS наследует все свойства класса заданий OLTP_JOBS. Классу заданий была сопоставлена группа потребителей ресурсов OLTP_GROUP, поэтому все параметры управления ресурсами, определенные для этой группы ресурсов, будут применяться к заданию COLLECT_STATS.

Управление окнами

Окно – это заданный интервал времени, в рамках которого выполняется определенное задание. При помощи окон можно контролировать выделение ресурсов (таких как процессор, серверы параллельных запросов, размер пула отката) и управлять ими. При выполнении задания активизируется метод выделения ресурсов, определенный активным (на текущий момент) окном, который и управляет выделением ресурсов для данного задания.

Окно состоит из трех основных компонентов:

- Расписание, которое определяет время начала действия окна.
- Продолжительность – время, в течение которого окно остается открытым.
- План выделения ресурсов, который применяется к заданию, связанному с окном.

Расписание может быть задано при помощи календарной строки или именованного расписания (оба эти способа были ранее рассмотрены для заданий). План выделения ресурсов можно задать через план диспетчера ресурсов, как это было описано в предыдущем разделе для

программ. В последующих разделах мы поговорим о втором компоненте окна – его продолжительности.

Создание окна

Для создания окна можно использовать процедуру CREATE_WINDOW пакета DBMS_SCHEDULER. Предположим, что необходимо создать окно LOW_LOAD, имеющее такие характеристики:

- Начинается каждый день в 3 часа утра ровно.
- Его продолжительность – 14 часов.
- Оно использует ресурсный план OLTP_PLAN.

Создадим такое окно.

```
 1  BEGIN
 2     DBMS_SCHEDULER.create_window (window_name    => 'low_load',
 3            resource_plan      => 'OLTP_PLAN',
 4            schedule_name      => 'SCHED_MANAGER.EVERY_DAY',
 5            DURATION           => NUMTODSINTERVAL
 6                                     (840,
 7                                      'minute'
 8                                      ),
 9            comments           => 'This is a low load window'
10            );
11 END;
```

Рассмотрим этот фрагмент кода подробно.

Строка	Описание
2	Имя окна.
3	Имя уже созданного плана диспетчера ресурсов, OLTP_PLAN.
4	Имя уже созданного расписания. Оно запускается ежедневно в 0:00.
5–8	Продолжительность окна. Значение для параметра duration должно иметь тип INTERVAL DAY TO SECOND. Четырнадцатичасовой интервал должен быть задан именно так, как в примере.
9	Комментарии для окна.

Создав окно, вы можете создавать задание, использующее это окно. Вызывая процедуру CREATE_JOB, можно указать имя окна в параметре schedule_name, как это сделано в последующем примере. Ранее говорилось о том, как задавать этот параметр для определения расписания. Теперь используем этот же параметр для задания окна LOW_LOAD.

```
BEGIN
   DBMS_SCHEDULER.create_job (job_name => 'Calculate_Daily_Interest',
                        program_name   => 'INTEREST_ADMIN.CALC_INTEREST',
                        schedule_name  => 'SYS.LOW_LOAD',
                        comments       => 'Daily Interest Calculation',
                        enabled        => TRUE
                             );
```

```
END;
```

Окна всегда принадлежат пользователю SYS, поэтому имя окна предваряется префиксом SYS. Задание связано с этим окном, поэтому метод выделения ресурсов для данного задания управляется схемой выделения ресурсов данного окна.

Когда окно активно, говорят, что оно *открыто*. Соответственно, когда окно становится неактивным, говорят, что оно *закрыто*. Окно открывается согласно сопоставленному ему календарю или расписанию.

Что происходит с заданиями, работающими в момент закрытия окна? Существует две возможности: вы можете позволить им выполняться дальше или остановить их. В некоторых случаях бывает важно остановить задание при закрытии окна. Пусть, например, у вас есть задание, собирающее статистику оптимизатора для таблиц. Это задание обычно выполняется ночью (возможно, до 6 часов утра). Предположим, что однажды оно не будет завершено в 6 часов утра и продолжит выполняться до 8 часов утра. В течение этого времени оно может оказывать влияние на обычную работу базы данных. В таком случае разумно остановить задание до завершения при закрытии соответствующего ему окна. Это свойство окна задается параметром stop_on_window_close процедуры CREATE_JOB. Если параметр установлен в значение TRUE, то при закрытии окна задание будет остановлено.

Любой пользователь, обладающий системной привилегией MANAGE_SCHEDULER, может создавать окна. Однако окна не будут принадлежать этому пользователю. С технической точки зрения объекты-окна принадлежат пользователю SYS.

Приоритеты окон

Окна определяются в соответствии со шкалой времени, поэтому в каждый момент времени активно только одно окно. Предположим, что вы определили два окна следующим образом:

- Окно W1 начинается в 7 часов утра и заканчивается в 9 часов вечера.
- Окно W2 начинается в 7 часов вечера и заканчивается в 7 часов утра.

Окна накладываются друг на друга в промежутке от 7 до 9 часов вечера. Какое из окон будет активно в течение этого промежутка времени?

Планировщик решает этот вопрос, анализируя присвоенные окнам приоритеты. Приоритеты определяются в параметре window_priority процедуры CREATE_WINDOW. Возможны всего два значения параметра: low (по умолчанию) и high, например:

```
window_priority => 'high'
```

При наложении окон будет активировано то, которое имеет приоритет high. При наложении окон с одинаковыми приоритетами более высо-

кий приоритет имеет окно, которое раньше закончится, второму придется подождать.

Указание даты окончания для окна

DBMS_SCHEDULER позволяет указать дату окончания для окна. Что это значит?

Предположим, что ваша бухгалтерская база данных получает информацию от внешнего источника. Вы обнаруживаете, что внешний источник дискредитирован и поступающие от него данные не являются достоверными. Вы решаете написать хранимую процедуру, которая будет исправлять информацию в базе данных и, в соответствии с расписанием, выполняться ежедневно. Известно, что подлежащие исправлению данные будут поступать только до 2 августа, а сегодня – 5 июля. После 2 августа вы хотите прекратить выполнение задания. Вы определяете окно так, чтобы задание не потребляло все имеющиеся ресурсы. Так как после 2 августа проблема будет снята, вы хотите ограничить продолжительность действия окна. При создании окна можно указать дату окончания его действия, завершить окно в указанный день и сэкономить ресурсы. Для этого используйте параметр end_date процедуры CREATE_WINDOW.

Получение информации об окнах

Сведения об окнах можно получить из представления базы данных DBA_SCHEDULER_WINDOWS. Рассмотрим наиболее важные столбцы этого представления.

Имя столбца	Описание
WINDOW_NAME	Имя окна.
RESOURCE_PLAN	Имя сопоставленного окну ресурсного плана.
SCHEDULE_OWNER	Если окну сопоставлено именованное расписание, то в данный столбец записывается имя владельца расписания.
SCHEDULE_NAME	Имя расписания (при его наличии).
START_DATE	Временная метка открытия окна. Действует только в случае, если календарные свойства окна задаются не через расписание, а через встроенную календарную строку. Отображает дату как значение типа TIMESTAMP(6) WITH TIME ZONE.
REPEAT_INTERVAL	Если для окна задано встроенное расписание (календарная строка, а не именованное расписание), то в данном столбце отображается соответствующая календарная строка.
END_DATE	Если для окна задано встроенное расписание (календарная строка, а не именованное расписание), то в данном столбце отображается временная метка окончательного закрытия окна. Аналогично столбцу start_date отображает дату как значение типа TIMESTAMP(6) WITH TIME ZONE.

Имя столбца	Описание
DURATION	Промежуток времени, в течение которого окно остается открытым (продолжительность). Отображается как значение типа DURATION.
WINDOW_PRIORITY	При перекрывании двух окон открывается окно с более высоким приоритетом (а окно с более низким приоритетом закрывается). В данном столбце отображается приоритет окна, который может иметь значение HIGH или LOW.
NEXT_START_DATE	Временная метка следующего по расписанию открытия окна. Отображает дату как значение типа TIMESTAMP(6) WITH TIME ZONE.
LAST_START_DATE	Временная метка последнего открытия окна. Отображает дату как значение типа TIMESTAMP(6) WITH TIME ZONE.
ENABLED	Указывает, включено ли окно. Может иметь значение TRUE или FALSE.
ACTIVE	Указывает, открыто ли окно в настоящий момент. Может иметь значение TRUE или FALSE.
COMMENTS	Комментарии для окна.

Удаление окон

Если в окне больше нет необходимости, его можно удалить при помощи процедуры DROP_WINDOW. Если вы попытаетесь удалить окно, которое используется в качестве расписания какими-то заданиями, выполняющимися в текущий момент времени, то обычным способом окно удалить не удастся. Придется использовать параметр FORCE следующим образом:

```
BEGIN
   DBMS_SCHEDULER.drop_window (window_name => 'window1', FORCE => TRUE);
END;
```

Если для окна задан параметр FORCE, то такое окно будет удалено вне зависимости от его связи с какими бы то ни было заданиями. В нашем примере при наличии каких-то заданий, использующих окно window1 как расписание, такие задания будут отключены, а окно – удалено.

Отключение и включение окон

Вместо того чтобы удалять окно, можно отключить его; после этого оно автоматически открываться не будет. Для отключения окна используйте процедуру DISABLE.

```
BEGIN
   DBMS_SCHEDULER.disable (NAME => 'window1');
END;
```

Если окно window1 в настоящий момент открыто, то выполнить эту операцию не удастся. Для принудительного отключения окна следует использовать параметр FORCE:

```
BEGIN
   DBMS_SCHEDULER.disable (NAME => 'window1', FORCE=>TRUE);
END;
```

Открытое в настоящий момент окно не закроется, но в будущем это окно автоматически не откроется.

Для включения отключенного окна используется процедура ENABLE:

```
BEGIN
   DBMS_SCHEDULER.enable (NAME => 'window1');
END;
```

Имейте в виду, что описанные здесь процедуры ENABLE и DISABLE — это те самые процедуры ENABLE/DISABLE, которые используются и для других объектов планировщика: заданий, расписаний и программ.

Принудительное открытие и закрытие окон

Давайте предположим, что для обычных OLTP-операций определено окно OLTP_WINDOW, начинающееся в 9 часов утра. Однако сегодня возникла нестандартная ситуация: обработка началась раньше, в 7 часов утра. Вы определили группу диспетчера ресурсов, связанную с окном, и хотите, чтобы задание по возможности выполнялось согласно данному ресурсному плану. Но окно будет открыто (как запланировано) только через два часа.

В подобном случае можно принудительно открыть окно при помощи процедуры OPEN_WINDOW:

```
BEGIN
   DBMS_SCHEDULER.open_window (
      window_name => 'OLTP_WINDOW', DURATION => NULL);
END;
```

Указанное окно будет открыто незамедлительно, и любые связанные с этим окном задания смогут сразу же использовать ресурсный план.

В нашем примере параметр duration установлен в NULL, что означает, что окно будет иметь свою обычную продолжительность. Предположим, что при создании окна была указано, что оно действует восемь часов, заканчиваясь в 5 часов вечера. Если окно открывается вручную в 7 часов утра, то автоматически оно будет закрыто через восемь часов после открытия (то есть в 3 часа дня). Возможно, для вас это неприемлемо и необходимо, чтобы окно в любом случае оставалось открытым до 5 часов вечера. Тогда при открытии окна можно явно указать необходимую продолжительность, указав значение типа INTERVAL.

```
BEGIN
   dbms_scheduler.open_window (
       window_name      => 'OLTP_WINDOW',
       DURATION         => NUMTODSINTERVAL (600, 'minute')
      );
END;
```

Группы окон

Давайте предположим, что были определены три разных окна:

MORNING

С 1:00 до 9:00 утра. В этом окне выполняются служебные задания, такие как сбор статистики.

WORKDAY

С 9:00 утра до 5:00 вечера. Все обычные OLTP-операции базы данных выполняются в этом окне. Выполнение пакетных и служебных заданий запрещено.

EVENING

С 5:00 вечера до 1:00 ночи. В этом окне работает большая часть пакетных заданий и ETL-операций.

Однако некоторые задания должны выполняться всегда, вне зависимости от того, какое окно активно в текущий момент времени (например, задание, следящее за производительностью). Еще одно окно определить нельзя, так как в каждый момент времени активным может быть только одно окно. Что же делать?

Можно определить *группу окон* (*window group*). При создании задания можно указать в качестве расписания группу окон, тогда задание будет наследовать свойства входящих в группу окон. Группа окон создается при помощи процедуры CREATE_WINDOW_GROUP. Например, для создания группы окон ALL_DAY, представляющей собой совокупность трех описанных ранее окон, пишем такой фрагмент кода:

```
BEGIN
   DBMS_SCHEDULER.create_window_group (
          group_name       => 'all_day',
          window_list      => 'MORNING, WORKDAY, EVENING',
          comments         => 'All day window group'
         );
END;
```

Имена всех окон, включаемых в группу, указываются через запятую в параметре window_list. Можно создать группу окон и не включая в нее ни одного окна. В этом случае параметр window_list устанавливается в значение NULL. Такая возможность удобна, если вы еще не решили, какие окна следует использовать.

При создании задания можно указать имя группы окон в параметре schedule_name процедуры CREATE_JOB:

```
BEGIN
   DBMS_SCHEDULER.create_job (job_name           => 'DB_Monitor',
                              program_name  => 'INTEREST_ADMIN.CALC_INTEREST',
                               schedule_name      => 'SYS.ALL_DAY',
                               comments           => 'DB Monitor',
                               enabled            => TRUE
                              );
```

```
END;
```

После того как группа окон создана, можно добавлять в нее окна и исключать их из группы. Для удаления окна MORNING из группы окон ALL_DAY используем процедуру REMOVE_WINDOW_GROUP_MEMBER:

```
BEGIN
   DBMS_SCHEDULER.remove_window_group_member
                            (group_name       => 'all_day',
                             window_list      => 'MORNING'
                            );
END;
```

Для добавления этого же окна обратно в группу используем процедуру ADD_WINDOW_GROUP_MEMBER:

```
BEGIN
   DBMS_SCHEDULER.add_window_group_member
                            (group_name       => 'all_day',
                             window_list      => 'MORNING'
                            );
END;
```

Для удаления группы окон используем процедуру DROP_WINDOW_GROUP:

```
BEGIN
   DBMS_SCHEDULER.drop_window_group (group_name => 'all_day');
END;
```

Если в группу уже включены какие-то окна, то удалить ее таким способом не удастся. В нашем примере в группу ALL_DAY включены три окна: MORNING, WORKDAY и EVENING. Для удаления такой группы используем параметр FORCE:

```
BEGIN
   DBMS_SCHEDULER.drop_window_group (group_name => 'all_day', FORCE => TRUE);
END;
```

Имейте в виду, что удалена будет только группа окон, но не входящие в нее окна: они останутся невредимыми.

Любые задания, использующие удаляемую группу окон, будут отключены. Однако их можно перезапустить вручную.

Управление журналированием

Как и большинство систем планирования, планировщик Oracle выполняет программы в фоновом режиме, так что непосредственная обратная связь с пользователями и администраторами отсутствует. Как же тогда выявить проблемы, связанные с выполнением заданий, после того как эти задания прекратили работать? Планировщик обеспечивает вас обширной журнальной информацией, порождаемой вашими операциями над заданиями и окнами. Журналирование обоих видов операций будет рассмотрено в последующих разделах.

Журналы заданий

Информацию о журналах заданий можно найти в двух представлениях словаря данных: DBA_SCHEDULER_JOB_LOG и DBA_SCHEDULER_JOB_RUN_DETAILS.

Представление DBA_SCHEDULER_JOB_LOG

При создании, изменении, удалении и запуске заданий результаты таких операций загружаются в сводную таблицу, которая доступна через представление DBA_SCHEDULER_JOB_LOG. Перечень столбцов представления приведен в таблице ниже.

Имя столбца	Описание
LOG_ID	Уникальный идентификатор записи.
LOG_DATE	Временная метка записи журнала.
OWNER	Владелец журнала.
JOB_NAME	Имя задания.
JOB_CLASS	Класс заданий (если задан).
OPERATION	Что произошло при обращении к заданию (например, CREATE, RUN, BROKEN).
STATUS	Что произошло после выполнения операции заданием (например, SUCCEEDED, FAILED).
USER_NAME	Имя пользователя, вызвавшего задание.
CLIENT_ID	Идентификатор клиента, если он был задан посредством DBMS_SESSION.SET_IDENTIFIER.
GLOBAL_UID	Глобальный UID для глобальных пользователей.
ADDITIONAL_INFO	Любая дополнительная информация о выполнении задания хранится в данном столбцев виде значения типа CLOB. Например, если задание было автоматически удалено после успешного выполнения (параметр auto_drop был установлен в TRUE), то в столбце OPERATION отображается значение DROP. Однако этих сведений недостаточно для того, чтобы понять, автоматически удалено задание или явно, пользователем. В столбце дополнительной информации выводится конкретная причина удаления, в данном случае это будет REASON="Auto drop job dropped". Аналогично здесь сохраняется и любая другая информация, связанная с выполнением задания.

Представление DBA_SCHEDULER_JOB_RUN_DETAILS

Представление DBA_SCHEDULER_JOB_LOG, описанное в предыдущем разделе, выводит сводную журнальную информацию для заданий, но не детали выполнения. Подробные сведения о выполнении работы содержатся в представлении DBA_SCHEDULER_JOB_RUN_DETAILS, столбцы которого описаны в следующей таблице.

Имя столбца	Описание
LOG_ID	Уникальный номер, представляющий определенную запись журнала.
LOG_DATE	Временная метка записи журнала, выводимая как значение типа TIMESTAMP WITH TIME ZONE.
OWNER	Владелец задания.
JOB_NAME	Имя задания.
STATUS	Статус задания после выполнения операции (например, FAILED, SUCCEEDED).
ERROR#	Номер ошибки Oracle (в случае ошибки).
REQ_START_DATE	Спланированное или запрошенное время запуска задания, которое может не совпадать с фактическим временем запуска.
ACTUAL_START_DATE	Выполнение задания может не начаться в запланированное или запрошенное время (например, если окно не было предварительно открыто или выполнялось более высокоприоритетное задание). В этом случае временная метка в данном столбце фиксирует фактическое время начала выполнения задания.
RUN_DURATION	Продолжительность выполнения задания, отображаемая как значение типа TIMESTAMP.
INSTANCE_ID	При работе с базой данных RAC задание должно выполняться на определенном экземпляре базы данных. Номер такого экземпляра записывается в данный столбец.
SESSION_ID	SID сеанса, который запустил процесс задания.
SLAVE_PID	Идентификатор подчиненного процесса задания.
CPU_USED	Процессорное время, израсходованное на выполнение задания.

ADDITIONAL_INFO	Дополнительная информация об особенностях выполнения задания, сохраненная в виде значения типа CLOB. Такая информация может быть чрезвычайно полезна для диагностики проблем с выполнением задания. Предположим, например, что задание выполнить не удалось, а в данном столбце находятся такие сведения:
	`ORA-01014: ORACLE shutdown in progress`
	Или предположим, что задание сбора статистики не выполнилось, и в столбце отображается следующее:
	`ORA-28031: maximum of 148 enabled roles exceeded`
	В обоих случаях вы можете определить точную причину невыполнения задания. Аналогично здесь сохраняется и другая информация, связанная с соответствующей строкой представления DETAILS.

Очистка журнала заданий

Журналирование заданий обеспечивает доступ к очень полезной информации, но если время от времени не очищать журналы, они будут неограниченно увеличиваться в объеме и заполнят все пространство базы данных. По умолчанию журнальная информация хранится в таблице (SYS.SCHEDULER$_JOB_RUN_DETAILS) табличного пространства SYSAUX. Oracle решает проблему, автоматически удаляя журнальные записи через определенные промежутки времени. Операция удаления выполняется заданием планировщика под названием PURGE_LOG, которое принадлежит пользователю SYS. Это задание автоматически устанавливается при создании базы данных Oracle в составе класса заданий DEFAULT_JOB_CLASS. Оно вызывает именованную программу PURGE_LOG_PROG, которая указывает на хранимую процедуру AUTO_PURGE самого пакета DBMS_SCHEDULER.

Это задание выполняется на основе именованного расписания DAILY_PURGE_SCHEDULE, которое запускается в 3:00 утра ежедневно. Хотя это автоматически создаваемое задание, это не мешает управлять его выполнением. Как и для любых других видов заданий, вы можете изменять его свойства (например, изменить время его запуска, вызываемую программу или определить, следует ли связывать его с окном).

PURGE_LOG удаляет только те записи журнала, которые помечены для удаления, так как истек их *период сохранения* (*retention period*) (см. далее раздел «Задание периода сохранения»).

Уровень журналирования

Управлять объемом журнала заданий можно также за счет ограничения объема информации о задании, записываемой в журнал. Например, можно вести журнал только при выполнении задания, но не при его создании. В некоторых случаях можно вообще отказаться от журналирования.

Определить уровень журналирования можно при создании класса заданий, задав параметр logging_level процедуры CREATE_JOB_CLASS. Впоследствии задание унаследует свойства класса (при наличии ссылки на класс при создании задания).

DBMS_SCHEDULER поддерживает три уровня журналирования:

DBMS_SCHEDULER.LOGGING_OFF

: Журналирование полностью отключено.

DBMS_SCHEDULER.LOGGING_RUNS

: Журналы ведутся только при выполнении задания. Значение по умолчанию.

DBMS_SCHEDULER.LOGGING_FULL

: Журналы ведутся при выполнении задания, а также при осуществлении любых операций над заданиями (таких как создание, удаление и изменение).

Можно задать параметр logging_level процедуры CREATE_JOB_CLASS следующим образом:

```
BEGIN
   DBMS_SCHEDULER.create_job_class
            (job_class_name      => 'MONITOR',
             logging_level       => dbms_scheduler.logging_full
             );
END;
```

Можно изменить уровень журналирования, задав атрибут logging_level класса заданий как в следующем примере (см. далее раздел «Управление атрибутами»).

```
BEGIN
   DBMS_SCHEDULER.set_attribute (NAME       => 'sys.default_job_class',
                                 ATTRIBUTE  => 'logging_level',
                                 VALUE      => dbms_scheduler.logging_full
                                  );
END;
```

Задание периода сохранения

Существует еще один способ ограничения размера журналов – задание периода сохранения для записей в журнальных таблицах. При запуске автоматического задания PURGE_LOG из журнала удаляются только те записи, которые старше соответствующей даты. Длительность периода сохранения по умолчанию составляет 30 дней. Через 30 дней журнальные записи, созданные 31 день назад, будут автоматически удалены.

При необходимости можно указать разные периоды сохранения для всех классов заданий. Для этого задайте параметр log_history процедуры CREATE_JOB_CLASS (значение по умолчанию – 30 дней). Позже можно будет изменить период сохранения для существующего класса заданий, задав атрибут log_history.

Зададим период сохранения для журнала продолжительностью 120 дней при создании класса заданий:

```
BEGIN
   DBMS_SCHEDULER.create_job_class (job_class_name      => 'MONITOR',
                                    log_history         => 120
                                   );
END;
```

В следующем примере период сохранения задается для уже существующего класса заданий DEFAULT_JOB_CLASS:

```
BEGIN
   DBMS_SCHEDULER.set_attribute (NAME          => 'sys.default_job_class',
                                 ATTRIBUTE     => 'log_history',
                                 VALUE         => 120
                                );
END;
```

Текущее значение длительности периода сохранения журнала можно найти в представлении словаря данных DBA_SCHEDULER_JOB_CLASSES:

```
SQL> SELECT job_class_name, logging_level, log_history
  2    FROM dba_scheduler_job_classes;

JOB_CLASS_NAME                 LOGG LOG_HISTORY
------------------------------ ---- -----------
DEFAULT_JOB_CLASS              RUNS         120
AUTO_TASKS_JOB_CLASS           RUNS
MONITOR                        RUNS         120
```

В большинстве случаев значения по умолчанию для уровня журналирования и периода сохранения оказываются вполне подходящими. Однако в некоторых случаях может возникнуть необходимость замены этих значений. Журналы обычно являются единственным средством, с помощью которого можно выявить проблемы после завершения заданий. Для особенно важных заданий можно включить полное журналирование. Для заданий, по которым вам необходимо просматривать более старые, чем обычно, сведения, можно указать более долгий (по отношению к значению по умолчанию) период сохранения.

Журналы окон

Как и для заданий, для окон пишутся журналы, которые доступны пользователю через два представления словаря данных, описанных в следующих разделах.

Возможность задания периода сохранения и уровня журналирования для журналов окон не поддерживается (в отличие от журналов заданий).

Представление DBA_SCHEDULER_WINDOW_LOG

При создании, изменении и удалении окон результаты операций загружаются в сводную таблицу, которая доступна пользователям через представление DBA_SCHEDULER_WINDOW_LOG. Перечень столбцов представления приведен в таблице ниже.

Имя столбца	Описание
LOG_ID	Уникальный идентификатор записи.
LOG_DATE	Временная метка записи журнала.
WINDOW_NAME	Имя окна.
OPERATION	Что произошло с окном при текущем обращении к нему (например, CLOSE, DISABLE, ENABLE, OPEN, UPDATE).
STATUS	Что произошло после выполнения операции заданием (например, SUCCEEDED, FAILED).
USER_NAME	Имя пользователя, вызвавшего задание.

Имя столбца	Описание
CLIENT_ID	Идентификатор клиента, если он был задан посредством DBMS_SESSION.SET_IDENTIFIER. Если такая команда была выдана из Oracle Enterprise Manager или Grid Control, то столбец заполняется следующим образом: `SYSMAN@192.168.1.1@Mozilla/4.0 (compatible; MSIE 6.0; Windows N` В этом примере мы видим, что пользователь Grid Control с именем SYSMAN вошел в систему с IP-адреса 192.168.1.1 и т. д.
GLOBAL_UID	Для глобального пользователя в данный столбец записывается глобальный UID.
ADDITIONAL_INFO	Любые дополнительные сведения хранятся в данном столбце в виде значения типа CLOB. Например, если программа была принудительно отключена, то столбец OPERATION содержит значение DISABLE, но не поясняет, как именно была отключена программа. Столбец ADDITIONAL_INFO выводит подробную информацию. Например, если окно было отключено вручную, то отображается следующее: `FORCE="TRUE", REASON="manually disabled"`

Представление DBA_SCHEDULER_WINDOW_DETAILS

Описанное в предыдущем разделе представление DBA_SCHEDULER_WINDOW_LOG содержит сводный журнал для окон, но не подробную информацию. Детали использования окон можно найти в представлении DBA_SCHEDULER_WINDOW_DETAILS, столбцы которого перечислены в таблице ниже.

Имя столбца	Описание
LOG_ID	Уникальный номер, представляющий определенную запись журнала.
LOG_DATE	Временная метка записи журнала, выводимая как значение типа TIMESTAMP WITH TIME ZONE.
WINDOW_NAME	Имя окна.
REQ_START_DATE	Запланированное или запрошенное время работы окна, которое может не совпадать с фактическим временем начала работы.
ACTUAL_START_DATE	Окно может не открыться или не закрыться в запланированное или запрошенное время. Это может произойти по ряду причин: окно не было включено, было открыто более высокоприоритетное окно или что-то еще. В этом случае в данном столбце отображается реальная временная метка начала окна.

Имя столбца	Описание
WINDOW_DURATION	Заданная продолжительность окна, отображаемая как значение типа INTERVAL. Может отличаться от реальной продолжительности окна.
ACTUAL_DURATION	Реальная продолжительность окна, отображаемая как значение типа INTERVAL. Может отличаться от заданной продолжительности окна.
INSTANCE_ID	Номер экземпляра для базы данных RAC.
ADDITIONAL_INFO	Любая дополнительная информация, сохраненная в виде значения типа CLOB. Такая информация может быть чрезвычайно полезна для диагностики проблем с окнами.

Управление атрибутами

На протяжении всей главы я показывал, как можно назначать свойства различным компонентам планировщика в момент их создания (например, посредством процедуры CREATE_JOB или CREATE_WINDOW). Однако в некоторых случаях требуется изменение таких свойств после создания компонента планировщика. Например, может возникнуть необходимость изменения интервала между выполнениями задания после того, как задание создано в базе данных. Можно изменить свойства, удалив и пересоздав объект, но это не единственное решение.

Такие свойства, как периодичность выполнения, имя расписания, уровень журналирования и другие, называют *атрибутами* задания, программы или объекта. Пакет DBMS_SCHEDULER содержит процедуру SET_ATTRIBUTE, которая позволяет изменять атрибуты существующего задания, класса заданий, программы, расписания, окна или группы окон.

Предположим, например, что необходимо изменить комментарий для задания PURGE_LOG на значение «This job purges the log entries of jobs and windows». Делаем следующее:

```
BEGIN
   DBMS_SCHEDULER.set_attribute
                       ('PURGE_LOG',
                        'comments',
                        'This job purges the log entries of jobs and windows'
                       );
END;
```

Процедура принимает три параметра:

имя_элемента

Имя элемента, свойства которого будут изменяться. Задания, окна, классы заданий, группы окон и расписания имеют уникальные имена, поэтому нет необходимости в указании типа элемента.

атрибут

Изменяемое свойство в виде значения типа VARCHAR2. Набор допустимых значений определяется типом элемента. Например, max_failures – допустимый атрибут для задания, но не для программы.

значение

Значение атрибута. Тип данных определяется выбором атрибута. Например, для окон атрибут window_priority принимает значение VARCHAR2 (LOW или HIGH), в то время как атрибут duration требует использования типа данных INTERVAL.

Теперь перейдем к рассмотрению допустимых значений для разных типов атрибутов. В связи с тем, что для каждого элемента они будут разными, каждый элемент будет рассмотрен в отдельном разделе. Атрибуты задаются явно и по одному, поэтому значений по умолчанию для них не предусмотрено. (Значения параметров по умолчанию были приведены при описании создания объектов.)

Задания

Атрибут	Тип данных	Описание
auto_drop	BOOLEAN	Указывает, следует ли удалять задание после его завершения. Допустимы следующие значения: TRUE и FALSE.
comments	VARCHAR2	Комментарии.
end_date	TIMESTAMP	Дата, после которой задание больше не будет запускаться. Задание будет удалено (если задан атрибут auto_drop) или отключено. Его статус изменится на COMPLETED (если оно завершено). Если этот атрибут задан, то атрибут schedule_name должен содержать NULL.
instance_stickiness	BOOLEAN	При работе в базе данных RAC задание запускается на любом одном экземпляре. Следующий запуск может быть осуществлен уже на другом экземпляре, который будет меньше загружен. Если данный атрибут установлен в значение TRUE, будет использоваться один и тот же экземпляр, вне зависимости от его загруженности. Допустимы следующие значения: TRUE и FALSE.
job_action	VARCHAR2	Природа программы. Для PL/SQL-блока указывается исполняемый анонимный блок. Для хранимой процедуры указывается ее имя (можно также указать имя схемы и пакета). Для исполняемого файла приводится полный путь к файлу операционной системы или сценарию оболочки.
job_class	VARCHAR2	Класс заданий, с которым связано данное задание.

Атрибут	Тип данных	Описание
job_priority	PLS_INTEGER	Приоритет здания по отношению к другим заданиям того же класса. Если выполнение нескольких заданий одного класса запланировано на одно и то же время, то порядок отбора заданий из класса для выполнения будет определяться по их приоритетам. Допустимы значения от 1 (наивысший приоритет) до 5 (самый низкий приоритет). Значение по умолчанию – 3.
job_type	VARCHAR2	Тип задания. Допустимы следующие значения: PLSQL_BLOCK STORED_PROCEDURE EXECUTABLE Если данный атрибут задан, то атрибут program_name должен содержать NULL.
job_weight	PLS_INTEGER	Степень параллелизма для задания. Допустимы значения от 1 до 100. Значение по умолчанию – 1.
logging_level	PLS_INTEGER	Подробность регистрируемой информации. Подменяет свойства класса заданий. Допустимы следующие значения: DBMS_SCHEDULER.LOGGING_OFF DBMS_SCHEDULER.LOGGING_RUNS (по умолчанию) DBMS_SCHEDULER.LOGGING_FULL
max_failures	PLS_INTEGER	Количество неудачных выполнений, по достижении которого статус задания изменится на BROKEN. Допустимы значения от 1 до 1000000. Значение по умолчанию – NULL, что означает, что новые экземпляры задания будут запускаться вне зависимости от того, сколько экземпляров уже не удалось выполнить. Отличие от старого пакета DBMS_JOB, в котором было разрешено максимум 16 неудачных выполнений задания, после чего оно помечалось как BROKEN (и эту цифру нельзя было изменить).
max_runs	NUMBER	Максимальное количество последовательных запланированных выполнений задания. При достижении этого максимума задание отключается, его статус меняется на COMPLETED. Допустимы значения от 1 до 1000000. Значение по умолчанию – NULL, что означает, что задание будет выполняться вечно или до достижения значений, указанных в параметре end_date или max_failures.
number_of_arguments	PLS_INTEGER	Количество аргументов для подставляемой (inline) программы. Если данный атрибут задан, то атрибут program_name должен содержать NULL.
program_name	VARCHAR2	Имя объекта-программы, который должен использоваться с этим заданием. Если данный атри-

Атрибут	Тип данных	Описание
		бут задан, то атрибуты job_action, job_type и number_of_arguments должны содержать NULL.
repeat_interval	INTERVAL	Символьное выражение, использующее синтаксис календарной строки (описанный ранее). Например: FREQ=YEARLY; BYMONTH=12.
restartable	BOOLEAN	Указывает, следует ли пытаться повторно выполнить задание после неудачного завершения. Допустимы следующие значения: TRUE и FALSE. Значение по умолчанию – TRUE.
schedule_limit	PLS_INTEGER	В сильнозагруженных системах задания не всегда запускаются в запланированное время. Этот атрибут указывает планировщику на то, что уже не следует запускать задание, если задержка оказалась больше указанного в атрибуте промежутка времени. Допустимы значения от 1 минуты до 99 дней. Например, если планировалось запустить задание в полдень и предельная задержка задана равной 60 минутам, то если задание не будет запущено до часа дня, оно не будет запущено уже никогда. Если schedule_limit не задан, то задание будет выполнено когда-нибудь впоследствии, когда освободятся ресурсы для его выполнения. По умолчанию атрибут установлен в NULL, что означает, что задание может быть запущено в любое время после запланированного. Задание, запуск которого пропущен согласно данному атрибуту, не учитывается в количестве успешных и неуспешных выполнений задания. В журнал вносится запись о пропуске выполнения задания.
schedule_name	VARCHAR2	Имя расписания, окна или группы окон, используемых в качестве расписания для данного задания. Если данный атрибут задан, то атрибуты end_date, start_date и repeat_interval должны (одновременно) содержать NULL.
start_date	VARCHAR2	Исходная временная метка начала выполнения задания или запланированного выполнения задания. Если данный атрибут задан, то атрибут schedule_name должен содержать NULL.
stop_on_window_close	BOOLEAN	Действие при закрытии окна. Что должно произойти с заданием, если оно связано с окном, а окно закрывается в процессе выполнения задания? Если данный атрибут установлен в значение TRUE, то задание также будет остановлено. Если атрибут установлен в значение FALSE, то выполнение задания будет продолжено даже при закрытии окна.

Классы заданий

Атрибут	Тип данных	Описание
comments	VARCHAR 2	Комментарии.
log_history	PLS_INTEGER	Срок хранения информации (в днях) в журнале для данного класса заданий. Допустимы значения от 1 до 999.
logging_level	PLS_INTEGER	Указывает уровень протоколирования. Допустимы следующие значения: DBMS_SCHEDULER.LOGGING_OFF DBMS_SCHEDULER.LOGGING_RUNS (по умолчанию) DBMS_SCHEDULER.LOGGING_FULL
resource_consumer_group	VARCHAR2	Группа потребителей ресурсов, с которой связывается данный класс заданий. Если этот атрибут задан, то атрибут service должен содержать NULL.
service	VARCHAR2	Имя сервиса (определенное в базе данных), которому принадлежит данный класс заданий. Значение по умолчанию NULL соответствует сервису по умолчанию. Если этот атрибут задан, то атрибут resource_consumer_group должен содержать NULL.

Расписания

Атрибут	Тип данных	Описание
comments	VARCHAR2	Комментарии.
end_date	TIMESTAMP	Временная метка отсечки, после которой расписание прекращает задание дат.
repeat_interval	VARCHAR2	Символьное выражение, использующее синтаксис календарной строки (описанный ранее). Например, FREQ=YEARLY; BYMONTH=12.
start_date	TIMESTAMP	Начальная временная метка, используемая в календарной строке.

Программы

Атрибут	Тип данных	Описание
comments	VARCHAR2	Комментарии.
number_of_arguments	PLS_INTEGER	Количество аргументов программы.
program_action	VARCHAR2	Природа программы: Для PL/SQL-блока указывается исполняемый анонимный блок. Для хранимой процедуры указывается ее имя (можно также указать имя схемы и пакета).

Атрибут	Тип данных	Описание
		Для исполняемого файла приводится полный путь к файлу операционной системы или сценарию оболочки.
program_type	VARCHAR2	Тип программы. Допустимы следующие значения: PLSQL_BLOCK STORED_PROCEDURE EXECUTABLE

Окна

Атрибут	Тип данных	Описание
comments	TIMESTAMP	Комментарии.
duration	INTERVAL	Продолжительность окна.
end_date	TIMESTAMP	Временная метка, после которой окно не будет открываться. Если данный атрибут задан, то атрибут schedule_name должен содержать NULL.
repeat_interval	VARCHAR2	Символьная строка, использующая синтаксис календарной строки (описанный ранее). Если данный атрибут задан, то атрибут schedule_name должен содержать NULL.
resource_plan	VARCHAR2	Ресурсный план, который связывается с окном.
schedule_name	TIMESTAMP	Имя расписания, которое будет использоваться для данного окна. Если этот атрибут задан, то атрибуты start_date, end_date и repeat_interval должны содержать NULL.
start_date	TIMESTAMP	Следующая временная метка, на которую запланировано открытие окна. Если этот атрибут задан, то атрибут schedule_name должен содержать NULL.
window_priority	VARCHAR2	Приоритет окна. Допустимы следующие значения: LOW и HIGH.

Группы окон

Атрибут	Тип данных	Описание
comments	TIMESTAMP	Комментарий для группы окон. Это единственный атрибут, поддерживаемый для группы окон.

Заключение

Планировщик Oracle Scheduler – это новая утилита управления заданиями, появившаяся в версии Oracle 10*g*. Она является гораздо более мощной, чем ее предшественник – пакет DBMS_JOB. При помощи планировщика можно обеспечивать выполнение хранимых процедур и анонимных блоков PL/SQL, а также исполняемых файлов операционной системы. Для календарных строк, задающих требуемое время выполнения заданий, используется нотация, практически не отличающаяся от обычного английского языка. Все операции планировщика доступны через программный интерфейс приложения во встроенном пакете DBMS_SCHEDULER. Кроме того, Enterprise Manager в Oracle 10*g* предоставляет графический интерфейс, который может быть использован для управления заданиями, что делает планирование выполнения заданий чрезвычайно простым даже для тех, кто пользуется им впервые. Планировщик позволяет определить именованное расписание, которое может вызываться независимо для выполнения действия, представляющего собой полное имя исполняемого файла или именованную программу, ссылающуюся на исполняемый файл. Задания также могут подчиняться системе управления ресурсами Oracle, которая может применяться для управления ресурсами (ЦПУ, серверы параллельных запросов), доступными отдельным заданиям. Подводя итог, можно сказать, что Scheduler – это единственная система управления заданиями, которая понадобится вам для планирования выполнения любых заданий, за исключением тех, которые точно должны выполняться вне связи с базой данных (например, для запуска самой базы данных).

A

Краткий справочник

В этом приложении приведены описания параметров и типов данных всех упомянутых в книге встроенных пакетов, а также перечислены столбцы представлений словаря данных, связанных с этими пакетами.

DBMS_OBFUSCATION_TOOLKIT

Пакет доступен в Oracle9*i* и последующих версиях, хотя и не рекомендован к применению в Oracle 10*g*. Он обеспечивает поддержку шифрования, генерации ключей и хеширования.

DES3GETKEY

Программа генерирует криптографически безопасный ключ шифрования. Ключ может использоваться в алгоритме шифрования DES3 (Triple Data Encryption Standard) как в двух-, так и в трехпроходном варианте. Программа реализована в виде перегруженных функции и процедуры, которые, в свою очередь, перегружены для различных типов данных.

Процедура (версия 1)

Принимает 2 входных параметра и возвращает ключ в выходном параметре.

Имя параметра	Тип данных	Описание
which	BINARY_INTEGER	Количество проходов в алгоритме Triple DES: 1 для двухпроходного, 2 для трехпроходного. По умолчанию 1 (двухпроходный).
seed_string	VARCHAR2	Строка начального значения для генерирования ключа.
key	VARCHAR2	Единственный выходной (OUT) параметр; в него помещается сгенерированный ключ.

Процедура (версия 2)

Идентична первой версии в том, что принимает два входных параметра и возвращает ключ в выходном параметре. Отличие в том, что параметры имеют тип RAW.

Имя параметра	Тип данных	Описание
which	BINARY_INTEGER	Количество проходов в алгоритме Triple DES: 1 для двухпроходного, 2 для трехпроходного. По умолчанию – 1 (двухпроходный).
seed	RAW	Строка начального значения для генерирования ключа.
key	RAW	Единственный выходной (OUT) параметр; в него помещается сгенерированный ключ.

Функция (версия 1)

Принимает два входных параметра и возвращает сгенерированный ключ. Возвращаемый ключ имеет тип VARCHAR2.

Имя параметра	Тип данных	Описание
which	BINARY_INTEGER	Количество проходов в алгоритме Triple DES: 1 для двухпроходного, 2 для трехпроходного. По умолчанию – 1 (двухпроходный).
seed_string	VARCHAR2	Строка начального значения для генерирования ключа.

Функция (версия 2)

Идентична первой версии за исключением того, что оперирует значениями типа RAW. Возвращает ключ в значении типа RAW.

Имя параметра	Тип данных	Описание
which	BINARY_INTEGER	Количество проходов в алгоритме Triple DES: 1 для двухпроходного, 2 для трехпроходного. По умолчанию 1 (двухпроходный).
seed_string	RAW	Строка начального значения для генерирования ключа.

DESGETKEY

Программа генерирует ключи для алгоритма DES (Data Encryption Standard). Как и DES3GETKEY, она реализована в виде двух перегруженных функций и двух процедур. Параметры имеют тот же смысл; отсутствует параметр which. (В алгоритме DES используется только один проход, поэтому нет необходимости в параметре, указывающем двух- или трехпроходный режим, как в случае DES3.)

DES3ENCRYPT

Программа применяется для зашифровывания входных данных по алгоритму DES3. Реализована в виде перегруженных функции и процедуры, которые, в свою очередь, перегружены для различных типов данных.

Процедура (версия 1)

Принимает четыре входных параметра и возвращает зашифрованное значение в выходном параметре.

Имя параметра	Тип данных	Описание
input_string	VARCHAR2	Входная строка, которая должна быть зашифрована. Ее длина должна быть кратна восьми.
key_string	VARCHAR2	Ключ шифрования. Его длина должна быть кратна восьми.
encrypted_string	VARCHAR2	Единственный выходной параметр; в нем передается зашифрованное значение.
which	BINARY_INTEGER	Количество проходов для алгоритма Triple DES: 1 для двух проходов, 2 для трех. По умолчанию – 1 (двухпроходный).
iv_string	VARCHAR2	Вектор инициализации. Это значение добавляется к входной строке для уменьшения повторяемости зашифрованных значений. Необязательный параметр. В случае использования суммарная длина входной строки и вектора инициализации должна быть кратна восьми.

Процедура (версия 2)

Идентична версии 1 за исключением того, что принимает параметры типа RAW.

Имя параметра	Тип данных	Описание
input	RAW	Входная строка, которая должна быть зашифрована.
key	RAW	Ключ шифрования.
encrypted	RAW	Единственный выходной параметр; в нем передается зашифрованное значение.
which	BINARY_INTEGER	Количество проходов для алгоритма Triple DES: 1 для двух проходов, 2 для трех. По умолчанию – 1 (двухпроходный).
iv	RAW	Вектор инициализации. Это значение добавляется к входной строке для уменьшения повторяемости зашифрованных значений. Необязательный параметр.

Функция (версия 1)

Идентична процедурной версии. Принимает четыре параметра и возвращает зашифрованное значение типа VARCHAR2.

Имя параметра	Тип данных	Описание
input_string	VARCHAR2	Входная строка, которая должна быть зашифрована. Ее длина должна быть кратна восьми.
key_string	VARCHAR2	Ключ шифрования. Его длина должна быть кратна восьми.
which	BINARY_INTEGER	Количество проходов для алгоритма Triple DES: 1 для двух проходов, 2 для трех. По умолчанию – 1 (двухпроходный).
iv_string	VARCHAR2	Вектор инициализации. Это значение добавляется к входной строке для уменьшения повторяемости зашифрованных значений. Необязательный параметр. В случае использования суммарная длина входной строки и вектора инициализации должна быть кратна восьми.

Функция (версия 2)

Идентична версии 1 за исключением того, что принимает параметры типа RAW и возвращает зашифрованное значение типа RAW.

Имя параметра	Тип данных	Описание
input	RAW	Входная строка, которая должна быть зашифрована.
key	RAW	Ключ шифрования.
which	BINARY_INTEGER	Количество проходов для алгоритма Triple DES: 1 для двух проходов, 2 для трех. По умолчанию – 1 (двухпроходный).
iv	RAW	Вектор инициализации. Это значение добавляется к входной строке для уменьшения повторяемости зашифрованных значений. Необязательный параметр.

DESENCRYPT

Программа зашифровывает данные по алгоритму DES. Перегружена двумя процедурами и двумя функциями, идентичными аналогичным программам DES3ENCRYPT; отсутствует параметр which. В алгоритме DES используется только один проход, поэтому нет необходимости в параметре, указывающем двух- или трехпроходный режим, как в случае DES3.

DES3DECRYPT

Программа расшифровывает зашифрованные данные по алгоритму DES3. Как и соответствующая ей DES3ENCRYPT, она реализована в виде перегруженных процедуры и функции, которые также перегружены для различных типов данных.

Процедура (версия 1)

Принимает четыре входных параметра и возвращает расшифрованное значение в выходном параметре.

Имя параметра	Тип данных	Описание
input_string	VARCHAR2	Зашифрованная строка, подлежащая расшифровыванию.
key_string	VARCHAR2	Ключ шифрования; тот же, который использовался для зашифровывания.
decrypted_string	VARCHAR2	Единственный выходной параметр; в нем передается расшифрованное значение.
which	BINARY_INTEGER	Количество проходов для алгоритма Triple DES: 1 для двух проходов, 2 для трех. По умолчанию – 1 (двухпроходный).
iv_string	VARCHAR2	Вектор инициализации. Это значение добавляется к входной строке для уменьшения повторяемости зашифрованных значений. Параметр необязателен, но если он был задан при зашифровывании, то при расшифровывании тоже должен быть указан.

Процедура (версия 2)

Идентична версии 1 за исключением того, что принимает параметры типа RAW.

Имя параметра	Тип данных	Описание
input	RAW	Зашифрованное значение, подлежащее расшифровыванию.
key	VARCHAR2	Ключ шифрования; тот же, который использовался для зашифровывания.
decrypted_data	VARCHAR2	Единственный выходной параметр; в нем передается расшифрованное значение.
which	BINARY_INTEGER	Количество проходов для алгоритма Triple DES: 1 для двух проходов, 2 для трех. По умолчанию – 1 (двухпроходный).
iv	VARCHAR2	Вектор инициализации. Это значение добавляется к входной строке для уменьшения

Имя параметра	Тип данных	Описание
		повторяемости зашифрованных значений. Параметр необязателен, но если он был задан при зашифровывании, то при расшифровывании тоже должен быть указан.

Функция (версия 1)

Идентична процедурной версии. Принимает четыре параметра и возвращает расшифрованное значение типа VARCHAR2.

Имя параметра	Тип данных	Описание
input_string	VARCHAR2	Зашифрованная строка, подлежащая расшифровыванию.
key_string	VARCHAR2	Ключ шифрования; тот же, который использовался для зашифровывания.
which	BINARY_INTEGER	Количество проходов для алгоритма Triple DES: 1 для двух проходов, 2 для трех. По умолчанию – 1 (двухпроходный).
iv_string	VARCHAR2	Вектор инициализации. Это значение добавляется к входной строке для уменьшения повторяемости зашифрованных значений. Параметр необязателен, но если он был задан при зашифровывании, то при расшифровывании тоже должен быть указан.

Функция (версия 2)

Идентична версии 1 за исключением того, что принимает параметры типа RAW и возвращает расшифрованное значение типа RAW.

Имя параметра	Тип данных	Описание
input_string	RAW	Зашифрованная строка, подлежащая расшифровыванию.
key_string	RAW	Ключ шифрования; тот же, который использовался для зашифровывания.
which	BINARY_INTEGER	Количество проходов для алгоритма Triple DES: 1 для двух проходов, 2 для трех. По умолчанию – 1 (двухпроходный).
iv_string	RAW	Вектор инициализации. Это значение добавляется к входной строке для уменьшения повторяемости зашифрованных значений. Параметр необязателен, но если он был задан при зашифровывании, то при расшифровывании тоже должен быть указан.

DESDECRYPT

Программа расшифровывает данные по алгоритму DES. Как и DES3DECRYPT, она перегружена двумя процедурами и двумя функциями, идентичными аналогичным программам DESENCRYPT; отсутствует параметр which. В алгоритме DES используется только один проход, поэтому нет необходимости в параметре, указывающем двух- или трехпроходный режим, как в случае DES3.

MD5

Программа используется для получения хеш-значения MD5 (Message Digest 5) для входного значения. Перегружена функцией и процедурой, которые также перегружены для различных типов данных.

Процедура (версия 1)

Принимает один входной параметр и возвращает хеш-значение в выходном параметре.

Имя параметра	Тип данных	Описание
input_string	VARCHAR2	Строка, для которой рассчитывается хеш-значение.
checksum_string	VARCHAR2	Выходной параметр, в котором возвращается хеш-значение.

Процедура (версия 2)

Идентична первой версии за исключением того, что работает с параметрами типа RAW.

Имя параметра	Тип данных	Описание
input	RAW	Значение, для которого рассчитывается хеш-значение.
checksum	RAW	Выходной параметр, в котором возвращается хеш-значение.

Функция (версия 1)

Принимает один входной параметр. Возвращает хеш-значение типа VARCHAR2 длиной 16 байт.

Имя параметра	Тип данных	Описание
input_string	VARCHAR2	Строка, для которой рассчитывается хеш-значение.

Функция (версия 2)

Принимает один входной параметр. Возвращает хеш-значение типа RAW длиной 16 байт.

Имя параметра	Тип данных	Описание
input	RAW	Значение, для которого рассчитывается хеш-значение.

DBMS_CRYPTO

Пакет доступен только в Oracle Database 10*g*. Как и DBMS_OBFUSCATION_TOOLKIT, используемый в Oracle9*i*, он включает в себя программы для зашифровывания, расшифровывания, генерации ключа и вычисления хеш-значений. В него также входят программы для работы с кодом аутентификации сообщения MAC.

GETRANDOMBYTES

Функция генерирует криптографически безопасный ключ шифрования. Принимает один входной параметр и возвращает ключ типа RAW.

Имя параметра	Тип данных	Описание
number_bytes	BINARY_INTEGER	Длина генерируемого случайного значения.

ENCRYPT

Программа зашифровывает полученное входное значение. Реализована в виде перегруженных функции и двух процедур, работающих с различными типами данных.

Функция

Принимает четыре входных параметра и возвращает зашифрованное значение типа RAW.

Имя параметра	Тип данных	Описание
src	RAW	Значение, которое должно быть зашифровано. Может иметь произвольную длину.
typ	BINARY_INTEGER	Комбинация алгоритма шифрования и методов дополнения и сцепления.
key	RAW	Ключ шифрования.
iv	RAW	Вектор инициализации. Это значение добавляется к входной строке для уменьшения повторяемости зашифрованных значений.

Процедура (версия 1)

Зашифровывает содержимое больших объектов LOB. Для шифрования значений других типов используйте функцию ENCRYPT. Эта версия принимает четыре входных параметра и возвращает зашифрованное значение в параметре типа BLOB.

Имя параметра	Тип данных	Описание
dst	BLOB	Выходной параметр; в нем передается зашифрованное значение.
src	BLOB	Значение типа BLOB или указатель ресурса, подлежащие зашифровыванию.
typ	BINARY_INTEGER	Комбинация алгоритма шифрования и методов дополнения и сцепления.
key	RAW	Ключ шифрования.
iv	RAW	Вектор инициализации. Это значение добавляется к входной строке для уменьшения повторяемости зашифрованных значений. Необязательный параметр.

Процедура (версия 2)

Идентична первой версии процедуры, за исключением того, что зашифровывает данные типа CLOB.

Имя параметра	Тип данных	Описание
dst	BLOB	Выходной параметр; в нем передается зашифрованное значение.
src	CLOB	Значение типа CLOB или указатель ресурса, подлежащие зашифровыванию.
typ	BINARY_INTEGER	Комбинация алгоритма шифрования и методов дополнения и сцепления.
key	RAW	Ключ шифрования.
iv	RAW	Вектор инициализации. Это значение добавляется к входной строке для уменьшения повторяемости зашифрованных значений. Необязательный параметр.

DECRYPT

Программа расшифровывает зашифрованные значения. Как и ENCRYPT, она реализована в виде перегруженных функции и двух процедур, работающих с разными типами данных.

Функция

Принимает четыре входных параметра и возвращает расшифрованное значение типа RAW.

Имя параметра	Тип данных	Описание
src	RAW	Зашифрованное значение, которое должно быть расшифровано.
typ	BINARY_INTEGER	Комбинация алгоритма шифрования и методов дополнения и сцепления. Должна совпадать с той, которая использовалась для зашифровывания.
key	RAW	Ключ шифрования; должен совпадать с тем, которым данные зашифровывались.
iv	RAW	Вектор инициализации. Это значение добавляется к входной строке для уменьшения повторяемости зашифрованных значений. Этот параметр должен быть указан, если он использовался при зашифровывании, и иметь такое же значение.

Процедура (версия 1)

Расшифровывает зашифрованные объекты LOB. Для расшифровывания данных других типов используйте функцию DECRYPT. Эта версия принимает четыре входных параметра и возвращает расшифрованное значение в параметре типа BLOB.

Имя параметра	Тип данных	Описание
dst	BLOB	Сюда помещается расшифрованное значение.
src	BLOB	Зашифрованное значение типа BLOB или указатель ресурса, которые должны быть расшифрованы.
typ	BINARY_INTEGER	Комбинация алгоритма шифрования и методов дополнения и сцепления. Должна совпадать с той, которая использовалась для зашифровывания.
key	RAW	Ключ шифрования; должен совпадать с тем, которым данные зашифровывались.
iv	RAW	Вектор инициализации. Это значение добавляется к входной строке для уменьшения повторяемости зашифрованных значений. Этот параметр должен быть указан, если он использовался при зашифровывании, и иметь такое же значение.

Процедура (версия 2)

Идентична первой версии процедуры за исключением того, что расшифровывает данные типа CLOB.

Имя параметра	Тип данных	Описание
dst	CLOB	Сюда помещается расшифрованное значение.
src	BLOB	Зашифрованное значение типа BLOB или указатель ресурса, которые должны быть расшифрованы.
typ	BINARY_INTEGER	Комбинация алгоритма шифрования и методов дополнения и сцепления. Должна совпадать с той, которая использовалась для зашифровывания.
key	RAW	Ключ шифрования; должен совпадать с тем, которым данные зашифровывались.
iv	RAW	Вектор инициализации. Это значение добавляется к входной строке для уменьшения повторяемости зашифрованных значений. Этот параметр должен быть указан, если он использовался при зашифровывании, и иметь такое же значение.

HASH

Программа генерирует криптографические хеш-значения для заданных входных данных. Вы можете создавать хеш-значения по алгоритму MD (Message Digest) или SHA-1 (Secure Hash Algorithm 1), задавая соответствующий параметр. Программа реализована тремя перегруженными функциями.

Функция (версия 1)

Генерирует хеш-значение для данных отличного от LOB типа. Эта версия принимает два параметра и возвращает хеш-значение типа RAW.

Имя параметра	Тип данных	Описание
src	RAW	Входное значение, для которого должно быть рассчитано хеш-значение.
typ	BINARY_INTEGER	Используемый алгоритм хеширования: DBMS_CRYPTO.HASH_MD5 для MD5 или DBMS_CRYPTO.HASH_SH1 для SHA-1.

Функция (версия 2)

Генерирует хеш-значение для данных типа BLOB. Эта версия принимает два параметра и возвращает хеш-значение типа RAW.

Имя параметра	Тип данных	Описание
src	BLOB	Входное значение типа BLOB или указатель ресурса, для которого должно быть рассчитано хеш-значение.

Имя параметра	Тип данных	Описание
typ	BINARY_INTEGER	Используемый алгоритм хеширования: DBMS_CRYPTO.HASH_MD5 для MD5 или DBMS_CRYPTO.HASH_SH1 для SHA-1.

Функция (версия 3)

Генерирует хеш-значение для данных типа CLOB. Эта версия принимает два параметра и возвращает хеш-значение типа RAW.

Имя параметра	Тип данных	Описание
src	CLOB	Входное значение типа CLOB или указатель ресурса, для которого должно быть рассчитано хеш-значение.
typ	BINARY_INTEGER	Используемый алгоритм хеширования: DBMS_CRYPTO.HASH_MD5 для MD5 или DBMS_CRYPTO.HASH_SH1 для SHA-1.

MAC

Программа генерирует код аутентификации сообщения MAC (Message Authentication Code) для указанного входного значения. Значения MAC аналогичны хеш-значениям, но в них добавлен ключ. Вы можете создавать MAC-значения по алгоритму MD (Message Digest) или SHA-1 (Secure Hash Algorithm 1), указывая соответствующее значение параметра typ. Как и HASH, эта программа реализована тремя перегруженными функциями.

Функция (версия 1)

Генерирует значение MAC для данных отличного от LOB типа. Эта версия принимает три параметра и возвращает значение MAC типа RAW.

Имя параметра	Тип данных	Описание
src	RAW	Входное значение, для которого должно быть рассчитано значение MAC.
typ	BINARY_INTEGER	Используемый алгоритм вычисления MAC: DBMS_CRYPTO.HMAC_MD5 для MD5 или DBMS_CRYPTO.HMAC_SH1 для SHA-1.
key	RAW	Ключ, используемый для вычисления значения MAC.

Функция (версия 2)

Генерирует значение MAC для данных типа BLOB. Эта версия принимает три параметра и возвращает значение MAC типа RAW.

Имя параметра	Тип данных	Описание
src	BLOB	Входное значение, для которого должно быть рассчитано значение MAC.
typ	BINARY_INTEGER	Используемый алгоритм вычисления MAC: DBMS_CRYPTO.HMAC_MD5 для MD5 или DBMS_CRYPTO.HMAC_SH1 для SHA-1.
key	RAW	Ключ, используемый для вычисления значения MAC.

Функция (версия 3)

Генерирует значение MAC для данных типа CLOB. Эта версия принимает три параметра и возвращает значение MAC типа RAW.

Имя параметра	Тип данных	Описание
src	CLOB	Входное значение, для которого должно быть рассчитано значение MAC.
typ	BINARY_INTEGER	Используемый алгоритм вычисления MAC: DBMS_CRYPTO.HMAC_MD5 для MD5 или DBMS_CRYPTO.HMAC_SH1 для SHA-1.
key	RAW	Ключ, используемый для вычисления значения MAC.

DBMS_RLS

Пакет включает в себя все программы, используемые для реализации безопасности на уровне строк, позволяя создавать, удалять, активировать, деактивировать и обновлять политики. В связи с тем, что в Oracle Database 10*g* некоторые параметры изменились и добавились новые, в таблицах отмечены различия между версиями Oracle9*i* и Oracle 10*g*.

ADD_POLICY

Процедура добавляет политику RLS для таблицы.

Имя параметра	Тип данных	Описание	Версия Oracle	
			9*i*	10*g*
object_schema	VARCHAR2	Владелец таблицы, к которой применяется политика RLS. По умолчанию – текущий пользователь.	Да	Да
object_name	VARCHAR2	Имя таблицы, к которой применяется политика RLS.	Да	Да
policy_name	VARCHAR2	Имя создаваемой политики RLS.	Да	Да
function_schema	VARCHAR2	Владелец функции политики. Функция формирует предикат, добавляемый к запросу для ограничения результирующего	Да	Да

Имя параметра	Тип данных	Описание	Версия Oracle	
			9*i*	10*g*
		множества. По умолчанию – текущий пользователь.	Да	Да
policy_function	VARCHAR2	Имя функции политики.	Да	Да
statement_types	VARCHAR2	Типы операторов, к которым применяется функция политики: SELECT, INSERT, UPDATE и/или DELETE. По умолчанию – все. В Oracle 10*g* Release 2 доступен новый тип INDEX.	Да	Да
update_check	BOOLEAN	Допустимы значения TRUE или FALSE. В случае TRUE при изменении/добавлении строк политика проверяет, что пользователь может видеть записи и после изменения. По умолчанию – FALSE.	Да	Да
enable	BOOLEAN	Допустимы значения TRUE или FALSE. Указывает, включена ли политика.	Да	Да
static_policy	BOOLEAN	Указывается для статической политики.	Да	Да
policy_type	BINARY_INTEGER	Динамизм политики: STATIC, SHARED_STATIC, CONTEXT_SENSITIVE, SHARED_CONTEXT_SENSITIVE или DYNAMIC. Дополняется префиксом DBMS_RLS, например POLICY_TYPE=> DBMS_RLS.STATIC. По умолчанию – DYNAMIC.	Нет	Да
long_predicate	BOOLEAN	Если длина возвращаемого функцией политики предиката превышает 4000 байт, необходимо передать данному параметру значение TRUE; это позволит функции возвращать предикаты длиной до 32000 байт. По умолчанию – FALSE.	Нет	Да
sec_relevant_cols	VARCHAR2	Указывает список столбцов, упоминание которых приводит к применению политики RLS; в противном случае к запросу политика не применяется.	Нет	Да
sec_relevant_cols_opt	VARCHAR2	Если имеются столбцы, при обращении к которым происходит применение политики RLS, то есть выбор: когда пользователь обращается к таким столбцам, следует ли показывать строки, подставляя в столбцы значение NULL, или не показывать эти строки вовсе? Передача этому параметру значения ALL_ROWS приводит к выбору первого варианта. Дополняется префиксом DBMS_RLS, например SEC_RELEVANT_ COLS_OPT => DBMS_RLS.ALL_ROWS. По умолчанию – NULL, что означает, что строки с такими значениями выводиться не будут.	Нет	Да

DROP_POLICY

Процедура отменяет политику RLS, назначенную таблице.

Имя параметра	Тип данных	Описание	Версия Oracle	
			9*i*	10*g*
object_schema	VARCHAR2	Владелец таблицы, к которой применяется политика RLS. По умолчанию – текущий пользователь.	Да	Да
object_name	VARCHAR2	Имя таблицы, к которой применяется политика RLS.	Да	Да
policy_name	VARCHAR2	Имя удаляемой политики RLS.	Да	Да

ENABLE_POLICY

Процедура включает и отключает политику RLS для таблицы.

Имя параметра	Тип данных	Описание	Версия Oracle	
			9*i*	10*g*
object_schema	VARCHAR2	Владелец таблицы, к которой применяется политика RLS. По умолчанию – текущий пользователь.	Да	Да
object_name	VARCHAR2	Имя таблицы, к которой применяется политика RLS.	Да	Да
policy_name	VARCHAR2	Имя политики RLS.	Да	Да
enable	BOOLEAN	Значение TRUE означает включение политики; FALSE – ее отключение.	Да	Да

REFRESH_POLICY

Процедура обновляет предикат политики RLS. Когда политика определена с типом, отличным от DYNAMIC, предикат политики может некоторое время не обновляться. Кэшированный в памяти предикат будет использоваться до тех пор, пока не выполнится определенное для него условие окончания срока действия. Если вы хотите обновить политику, то просто выполните процедуру REFRESH_POLICY. Она заново выполнит функцию политики и обновит в кэше предикат.

Имя параметра	Тип данных	Описание	Версия Oracle	
			9*i*	10*g*
object_schema	VARCHAR2	Владелец таблицы, к которой применяется политика RLS. По умолчанию – текущий пользователь.	Да	Да

Имя параметра	Тип данных	Описание	Версия Oracle	
			9*i*	10*g*
object_name	VARCHAR2	Имя таблицы, к которой применяется политика RLS.	Да	Да
policy_name	VARCHAR2	Имя обновляемой политики RLS.	Да	Да

Представления словаря данных для RLS

В этом разделе описаны столбцы представлений словаря данных, имеющие отношение к безопасности на уровне строк.

DBA_POLICIES

В этом представлении отображаются все имеющиеся в базе данных политики RLS независимо от того, включены они или нет.

Имя параметра	Описание	Версия Oracle	
		9*i*	10*g*
OBJECT_OWNER	Владелец таблицы, для которой определена данная политика.	Да	Да
OBJECT_NAME	Имя таблицы, для которой определена данная политика.	Да	Да
POLICY_GROUP	Если политика входит в группу, то здесь указывается имя группы.	Да	Да
POLICY_NAME	Имя политики.	Да	Да
PF_OWNER	Владелец функции политики, формирующей и возвращающей предикат.	Да	Да
PACKAGE	Если функция политики входит в пакет, то здесь указывается имя пакета.	Да	Да
FUNCTION	Имя функции политики.	Да	Да
SEL	Признак применения политики к операторам SELECT для данной таблицы.	Да	Да
INS	Признак применения политики к операторам INSERT для данной таблицы.	Да	Да
UPD	Признак применения политики к операторам UPDATE для данной таблицы.	Да	Да
DEL	Признак применения политики к операторам DELETE для данной таблицы.	Да	Да
IDX	Признак применения политики к операторам CREATE INDEX для данной таблицы (только для Oracle Database 10*g* Release 2).	Нет	Да

Имя параметра	Описание	Версия Oracle	
		9*i*	10*g*
CHK_OPTION	Признак включения режима update_check при создании политики.	Да	Да
ENABLE	Признак включенного состояния политики.	Да	Да
STATIC_POLICY	Признак того, является ли политика статической.	Да	Да
POLICY_TYPE	Тип динамизма политики (например, STATIC). Полный список см. в ADD_POLICY.	Нет	Да
LONG_PREDICATE	Признак того, возвращает ли политика предикат длиннее, чем 4000 байт.	Нет	Да

DBMS_FGA

Пакет применяется для добавления, удаления, включения и отключения политик детального аудита. В связи с тем, что некоторые параметры появились или имеют другое значение в Oracle Database 10*g*, в приведенных таблицах отмечены различия версий Oracle9*i* Database и Oracle Database 10*g*.

ADD_POLICY

Процедура добавляет политику FGA для таблицы.

Имя параметра	Тип данных	Описание	Версия Oracle	
			9*i*	10*g*
object_schema	VARCHAR2	Имя схемы, к таблицам которой применяется FGA.	Да	Да
object_name	VARCHAR2	Имя таблицы, к которой применяется FGA.	Да	Да
policy_name	VARCHAR2	Имя политики.	Да	Да
audit_condition	VARCHAR2	Условие, при котором будет формироваться журнал аудита (например, USER= 'SCOTT').	Да	Да
audit_column	VARCHAR2	Аудит будет срабатывать только при обращении к столбцам из этого списка.	Да	Да
handler_schema	VARCHAR2	Владелец модуля обработки политики, если имеется такой модуль. Модули обработки – это процедуры или пакеты, автоматически выполняющиеся при наступлении условия аудита.	Да	Да
handler_module	VARCHAR2	Имя процедуры или пакета модуля обработки.	Да	Да

Имя параметра	Тип данных	Описание	Версия Oracle	
			9*i*	10*g*
enable	BOOLEAN	Признак того, что политика включается при создании. По умолчанию включается.	Да	Да
statement_types	VARCHAR2	Типы операторов, подлежащие аудиту. Возможные значения: SELECT, INSERT, DELETE и UPDATE.	Нет	Да
audit_trail	BINARY_INTEGER	Если требуется фиксировать наряду с SQL-текстом переменные связывания, укажите в этом параметре DB_EXTENDED (значение по умолчанию). Иначе укажите DB. В Oracle 10*g* Release 2 можно указать еще один тип – XML, при котором журнал аудита пишется в файловую систему в формате XML.	Нет	Да
audit_column_opts	BINARY_INTEGER	Если указан параметр audit_column, то аудит включается только при обращении к указанным в нем столбцам. Если данному параметру присвоено значение ALL_COLUMNS, то аудит срабатывает только при обращении ко всем столбцам параметра audit_column одновременно. Если данный параметр имеет значение ANY_COLUMNS, то аудит срабатывает при обращении к любому из указанных столбцов.	Нет	Да

DROP_POLICY

Процедура удаляет политику FGA, ранее назначенную таблице.

Имя параметра	Тип данных	Описание	Версия Oracle	
			9*i*	10*g*
object_schema	VARCHAR2	Владелец таблицы, к которой применяется FGA.	Да	Да
object_name	VARCHAR2	Имя таблицы, к которой применяется FGA.	Да	Да
policy_name	VARCHAR2	Имя удаляемой политики.	Да	Да

DISABLE_POLICY

Процедура отключает политику FGA, ранее созданную для таблицы. Сама политика не удаляется, но перестает действовать.

Имя параметра	Тип данных	Описание	Версия Oracle	
			9*i*	10*g*
object_schema	VARCHAR2	Владелец таблицы, к которой применяется FGA.	Да	Да
object_name	VARCHAR2	Имя таблицы, к которой применяется FGA.	Да	Да
policy_name	VARCHAR2	Имя отключаемой политики.	Да	Да

ENABLE_POLICY

Процедура включает политику FGA, ранее созданную для таблицы. Включаемая политика должна быть создана заранее.

Имя параметра	Тип данных	Описание	Версия Oracle	
			9*i*	10*g*
object_schema	VARCHAR2	Владелец таблицы, к которой применяется FGA.	Да	Да
object_name	VARCHAR2	Имя таблицы, к которой применяется FGA.	Да	Да
policy_name	VARCHAR2	Имя включаемой политики.	Да	Да

Представления словаря данных для FGA

В этом разделе описаны столбцы представлений словаря данных, имеющие отношение к детальному аудиту.

DBA_AUDIT_POLICIES

Представление отображает сведения об имеющихся в базе данных политиках FGA.

Имя параметра	Описание	Версия Oracle	
		9*i*	10*g*
OBJECT_SCHEMA	Владелец таблицы, для которой определена политика.	Да	Да
OBJECT_NAME	Имя таблицы, для которой определена политика.	Да	Да
POLICY_NAME	Имя определяемой политики.	Да	Да
POLICY_TEXT	Условие, при котором должен включаться аудит (например, SALARY>1500).	Да	Да
POLICY_COLUMN	Столбцы, обращение к которым активирует аудит.	Да	Да
PF_SCHEMA	Если у политики есть модуль обработки, здесь указывается его владелец. Модуль обработки автоматически выполняется при наступлении условия аудита.	Да	Да

Имя параметра	Описание	Версия Oracle	
		9*i*	10*g*
PF_PACKAGE	Если модуль обработки входит в состав пакета, здесь указывается имя этого пакета.	Да	Да
PF_FUNCTION	Имя модуля обработки. Если этот модуль является процедурой в составе пакета, здесь указывается имя процедуры.	Да	Да
ENABLED	Признак того, включена ли в данный момент политика.	Да	Да
SEL	Признак применения политики к операторам SELECT.	Нет	Да
INS	Признак применения политики к операторам INSERT.	Нет	Да
UPD	Признак применения политики к операторам UPDATE.	Нет	Да
DEL	Признак применения политики к операторам DELETE.	Нет	Да
AUDIT_TRAIL	Тип аудита. Если указано значение DB_EXTENDED, то в журнал записываются переменные связывания. Если указано DB, то переменные связывания не записываются. В Oracle 10*g* Release 2 может иметь значение XML, что означает запись журнала в файловую систему в формате XML.	Нет	Да
POLICY_COLUMN_OPTIONS	Признак активации аудита при обращении ко всем выбранным столбцам или к любому из столбцов, перечисленных в столбце POLICY_COLUMN.	Нет	Да

DBA_FGA_AUDIT_TRAIL

Представление отображает журнал аудита FGA.

Имя параметра	Описание	Версия Oracle	
		9*i*	10*g*
SESSION_ID	Идентификатор AUDIT SESSION. Уникальный номер сеанса; не связан со значением столбца SID в представлении V$SESSION.	Да	Да
TIMESTAMP	Время записи в журнал.	Да	Да
DB_USER	Пользователь БД, выполнивший этот зарегистрированный в журнале запрос.	Да	Да
OS_USER	Пользователь операционной системы.	Да	Да
USERHOST	Имя хоста пользователя.	Да	Да
CLIENT_ID	Идентификатор клиента в данном сеансе, если определен.	Да	Да
ECONTEXT_ID	Только для Oracle 10*g* Release 2. Если контекст определен, здесь находится идентификатор контекста.	Нет	Да (R2)

Имя параметра	Описание	Версия Oracle	
		9*i*	10*g*
EXT_NAME	Если пользователь использует внешнюю аутентификацию, то здесь отображается внешнее имя.	Да	Да
OBJECT_SCHEMA	Владелец таблицы, к которой обращается оператор.	Да	Да
OBJECT_NAME	Имя таблицы.	Да	Да
POLICY_NAME	Имя политики, породившей данную запись.	Да	Да
SCN	Номер SCN на момент создания данной журнальной записи. Используется в ретроспективных запросах.	Да	Да
SQL_TEXT	Текст оператора SQL, выполненного пользователем.	Да	Да
SQL_BIND	Переменные связывания и их значения в операторе SQL.	Да	Да
COMMENT$TEXT	Дополнительные данные аудита, если таковые имеются.	Да	Да
STATEMENT_TYPE	Тип оператора (например, SELECT).	Нет	Да
EXTENDED_TIMESTAMP	Расширенная временная метка записи. Данные типа TIMESTAMP с точностью до одной тысячной секунды.	Нет	Да
PROXY_SESSIONID	Если пользователь использует доверенное соединение, здесь отображается SID прокси-сеанса.	Нет	Да
GLOBAL_UID	Для корпоративных пользователей, аутентифицированных через LDAP или иной механизм, здесь указывается глобальный идентификатор UID.	Нет	Да
INSTANCE_NUMBER	Идентификатор экземпляра, относится только к конфигурации Real Application Cluster (RAC).	Нет	Да
OS_PROCESS	Идентификатор процесса операционной системы.	Нет	Да
TRANSACTIONID	Если оператор представляет собой транзакцию, здесь отображается ее идентификатор.	Нет	Да
STATEMENTID	В одном сеансе может быть несколько операторов (например, когда пользователь делает выборку из одной таблицы, затем из другой и т. д.). Каждый оператор получает собственный идентификатор.	Нет	Да
ENTRYID	Один оператор может вызываться рекурсивно; каждому вхождению соответствует собственный уникальный идентификатор.	Нет	Да

FLASHBACK_TRANSACTION_QUERY

Представление отображает сведения о транзакциях, имеющихся в сегменте отката системы. Можете использовать эту информацию для выяснения деталей транзакции. Доступно только в Oracle 10*g*.

Имя столбца	Описание
XID	Идентификатор транзакции.
START_SCN	Номер SCN на момент начала транзакции.
START_TIMESTAMP	Время начала транзакции в формате TIMESTAMP.
COMMIT_SCN	Номер SCN на момент фиксации транзакции.
COMMIT_TIMESTAMP	Время фиксации транзакции в формате TIMESTAMP.
LOGON_USER	Пользователь, выполнивший транзакцию.
UNDO_CHANGE#	Номер SCN для операции отката.
OPERATION	Операция, выполненная транзакцией (например, SELECT). Блоки PL/SQL будут показаны как DECLARE или BEGIN.
TABLE_NAME	Имя таблицы, над которой была выполнена операция.
TABLE_OWNER	Владелец таблицы.
ROW_ID	Значение ROWID для строки, модифицированной данной транзакцией.
UNDO_SQL	Оператор SQL, который может быть использован для отката транзакции.

DBMS_RANDOM

Пакет содержит программы, генерирующие случайные значения (числовые и строковые) и выполняющие другие операции, связанные с рандомизацией.

SEED

Программа инициализирует пакет DBMS_RANDOM указанным пользователем начальным значением, которое затем используется при генерировании случайных значений. Использование в разных программах различных начальных значений позволяет достичь достаточной степени случайности. Реализация SEED включает две перегруженные функции, принимающие параметры разного типа.

Функция (версия 1)

Принимает единственный параметр val, используемый в качестве начального значения. Параметр имеет тип VARCHAR2 и может содержать до 2000 байт.

Функция (версия 2)

Принимает единственный параметр val, используемый в качестве начального значения. Параметр имеет тип BINARY_INTEGER.

VALUE

Программа возвращает случайное положительное число с плавающей точкой, меньшее единицы. Реализация VALUE включает две перегруженные функции, одна из которых принимает параметры, а другая – нет.

Функция без параметров (версия 1)

Возвращает случайное число типа NUMBER. Параметров нет.

Функция с параметрами (версия 2)

Возвращает случайное число типа NUMBER. Принимает два входных параметра: low и high (нижнее и верхнее значения). Возвращаемое значение находится в указанном диапазоне.

Имя параметра	Тип данных	Описание
low	NUMBER	Нижняя граница для возвращаемого случайного числа.
high	NUMBER	Верхняя граница для возвращаемого случайного числа.

STRING

Функция возвращает случайную строку. Длина строки и набор символов определяются пользователем. Функция принимает два входных параметра.

Имя параметра	Тип данных	Описание
opt	VARCHAR2	Параметр, определяющий набор символов генерируемой строки (допустимые значения приведены в следующей таблице).
len	NUMBER	Длина генерируемой случайной строки.

Параметр opt может принимать любое из перечисленных в следующей таблице значений.

Значение	Действие
u	Использовать только буквы верхнего регистра (например, DFTHNDSW).
l	Использовать только буквы нижнего регистра (например, pikdcdsd).
a	Использовать буквы обоих регистров (например, DeCWCass).
x	Использовать совместно буквы верхнего регистра и цифры (например, A1W56RTY).
p	Использовать любые печатные символы (например, $\$2sw&*).

NORMAL

Функция NORMAL, как и VALUE, возвращает случайное число. Однако генерируемые ею значения подчиняются нормальному закону распределения. Это означает, что после серии успешных выполнений сгенерированные значения будут иметь нормальное распределение. Параметры отсутствуют.

DBMS_SCHEDULER

Пакет применяется для запуска фоновых заданий, выполняемых в определенный момент времени или с заданной периодичностью. В него входят все программы, необходимые для создания и обслуживания заданий, классов заданий, расписаний, программ, окон и групп окон.

CREATE_JOB

Программа создает задание, которое должно быть выполнено в данный момент либо позже. Реализована в виде четырех перегруженных процедур.

Процедура без именованных объектов (версия 1)

Эта версия позволяет создать задание наиболее быстрым способом. Принимает в качестве параметров все необходимые для создания задания компоненты (календарь, описание действия и т. п.), определенные по отдельности, без явного их именования.

Имя параметра	Тип данных	Описание
job_name	VARCHAR2	Имя задания.
job_type	VARCHAR2	Тип задания. Корректными значениями являются PLSQL_BLOCK, STORED_PROCEDURE и EXECUTABLE.
job_action	VARCHAR2	Фактическое содержание задания. Зависит от предыдущего параметра. Если тип задания – PLSQL_BLOCK, то это полностью весь PL/SQL-блок. Тип STORED_PROCEDURE требует имени хранимой процедуры. Тип EXECUTABLE требует имени исполняемого файла ОС с указанием полного пути.
number_of_arguments	PLS_INTEGER	Если программа, определенная в предыдущем параметре, принимает входные параметры, то здесь указывается их количество. По умолчанию 0.
start_date	TIMESTAMP WITH TIMEZONE	Дата и время запланированного запуска задания. Значение по умолчанию NULL, что означает, что задание запускается немедленно.
repeat_interval	VARCHAR2	Календарная строка, определяющая сроки запуска задания, например «FREQ=DAILY BY HOUR =3». Полный список допустимых значений

Имя параметра	Тип данных	Описание
		приведен в главе 8. Значение по умолчанию NULL, что означает однократное выполнение задания в момент start_date без повторных запусков.
end_date	TIMESTAMP WITH TIME-ZONE	Если задано значение, то оно определяет момент окончания выполнения задания. Значение по умолчанию NULL, что означает бесконечное выполнение.
job_class	VARCHAR2	Классы заданий определяют назначенный заданию профиль потребления ресурсов, уровень журналирования и т. п. Oracle поставляется с предопределенным классом заданий DEFAULT_JOB_CLASS. Если этот параметр не задан, то задание относится к данному классу по умолчанию.
enabled	BOOLEAN	Указывает, включено ли задание. Значение по умолчанию – FALSE, которое означает, что задание создается отключенным. Чтобы включить его при создании, следует указать в этом параметре значение TRUE.
auto_drop	BOOLEAN	Если значение параметра TRUE, то задание удаляется после выполнения. Значение FALSE отменяет удаление. По умолчанию – TRUE.
comments	VARCHAR2	Комментарии к заданию.

Процедура с именованным расписанием и подставляемой программой (версия 2)

Идентична версии 1 за исключением того, что вместо календарной строки принимает именованное расписание.

Имя параметра	Тип данных	Описание
job_name	VARCHAR2	Имя задания.
schedule_name	VARCHAR2	Имя ранее созданного расписания. Если владельцем расписания является другой пользователь, то его имя указывается в качестве префикса (например, SCHED_ADMIN.EVERY_DAY). Т. к. расписание определяет моменты запуска, останова и периодичность выполнения задания, соответствующие параметры (start_date, end_date и repeat_interval) отсутствуют.
job_type	VARCHAR2	Тип задания. Корректными значениями являются PLSQL_BLOCK, STORED_PROCEDURE и EXECUTABLE.
job_action	VARCHAR2	Фактическое содержание задания. Зависит от предыдущего параметра. Если тип задания – PLSQL_BLOCK, то это полностью весь PL/SQL-блок.

Имя параметра	Тип данных	Описание
		Тип STORED_PROCEDURE требует имени хранимой процедуры. Тип EXECUTABLE требует имени исполняемого файла ОС с указанием полного пути.
number_of_arguments	PLS_INTEGER	Если программа, определенная в предыдущем параметре, принимает входные параметры, то здесь указывается их количество. По умолчанию – 0.
job_class	VARCHAR2	Классы заданий определяют назначенный заданию профиль потребления ресурсов, уровень журналирования и т. п. Oracle поставляется с предопределенным классом заданий DEFAULT_JOB_CLASS. Если данный параметр не задан, то задание относится к классу по умолчанию.
enabled	BOOLEAN	Указывает, включено ли задание. Значение по умолчанию – FALSE, которое означает, что задание создается отключенным. Чтобы включить его при создании, следует указать в этом параметре TRUE.
auto_drop	BOOLEAN	Если значение параметра TRUE, то задание удаляется после выполнения. Значение FALSE отменяет удаление. По умолчанию – TRUE.
comments	VARCHAR2	Комментарии к заданию.

Процедура с именованной программой и подставляемым расписанием (версия 3)

Идентична версии 2 за исключением того, что принимает календарную строку и именованную программу. В этой версии именованная программа используется вместо явно определяемого действия.

Имя параметра	Тип данных	Описание
job_name	VARCHAR2	Имя задания.
program_name	VARCHAR2	Имя определенной вами ранее именованной программы. Если программа принадлежит другому пользователю, его имя следует использовать в качестве префикса (например, JOB_ADMIN.CALC_INT). Т. к. программа определяет все характеристики выполняемого действия, соответствующие параметры (в частности, job_action) отсутствуют.
start_date	TIMESTAMP WITH TIMEZONE	Дата и время запланированного запуска задания. Значение NULL (значение по умолчанию) означает, что задание запускается немедленно.
repeat_interval	VARCHAR2	Календарная строка, определяющая сроки запуска задания, например, «FREQ=DAILY BY

Имя параметра	Тип данных	Описание
		HOUR=3». Полный список допустимых значений приведен в главе 8. Значение по умолчанию NULL, что означает однократное выполнение задания в момент start_date без повторных запусков.
end_date	TIMESTAMP WITH TIME-ZONE	Если задано значение, то оно определяет момент окончания выполнения задания. Значение по умолчанию NULL, что означает бесконечное выполнение.
job_class	VARCHAR2	Классы заданий определяют назначенный заданию профиль потребления ресурсов, уровень журналирования и т. п. Oracle поставляется с предопределенным классом заданий DEFAULT_JOB_CLASS. Если данный параметр не задан, то задание относится к классу по умолчанию.
enabled	BOOLEAN	Указывает, включено ли задание. Значение по умолчанию – FALSE, которое означает, что задание создается отключенным. Чтобы включить его при создании, следует указать в этом параметре TRUE.
auto_drop	BOOLEAN	Если значение параметра TRUE, то задание удаляется после выполнения. Значение FALSE отменяет удаление. По умолчанию – TRUE.
comments	VARCHAR2	Комментарии к заданию.

Процедура с именованной программой и именованным расписанием (версия 4)

Самая простая из всех процедур создания заданий. Если вы создаете задание этой версией процедуры, вам надо указать только имя программы и имя расписания, которые должны быть определены заранее.

Имя параметра	Тип данных	Описание
job_name	VARCHAR2	Имя задания.
program_name	VARCHAR2	Имя определенной вами ранее именованной программы. Если программа принадлежит другому пользователю, его имя следует использовать в качестве префикса (например, JOB_ADMIN.CALC_INT). Т. к. программа определяет все характеристики выполняемого действия, соответствующие параметры (в частности, job_action) отсутствуют.
schedule_name	VARCHAR2	Имя ранее созданного расписания. Если владельцем расписания является другой пользователь, то его имя указывается в качестве префикса (например, SCHED_ADMIN.EVERY_DAY). Т. к. расписание определяет моменты запуска, останова

Имя параметра	Тип данных	Описание
		и периодичность выполнения задания, соответствующие параметры (start_date, end_date и repeat_interval) отсутствуют.
job_class	VARCHAR2	Классы заданий определяют назначенный заданию профиль потребления ресурсов, уровень журналирования и т. п. Oracle поставляется с предопределенным классом заданий DEFAULT_JOB_CLASS. Если данный параметр не задан, то задание относится к классу по умолчанию.
enabled	BOOLEAN	Указывает, включено ли задание. Значение по умолчанию – FALSE, которое означает, что задание создается отключенным. Чтобы включить его при создании, следует указать в этом параметре TRUE.
auto_drop	BOOLEAN	Если значение параметра TRUE, то задание удаляется после выполнения. Значение FALSE отменяет удаление. По умолчанию – TRUE.
comments	VARCHAR2	Комментарии к заданию.

CREATE_JOB_CLASS

Процедура создает класс заданий, который в дальнейшем может быть сопоставлен заданиям. Класс может не содержать в себе заданий. Когда задания включаются в класс, им присваиваются его атрибуты (уровень журналирования, группа потребителей ресурсов и т. п.).

Имя параметра	Тип данных	Описание
job_class_name	VARCHAR2	Имя класса заданий. Должно быть уникально для базы данных.
resource_consumer_group	VARCHAR2	Имя группы потребителей ресурсов, определенной в базе данных ранее. Значение по умолчанию отсутствует. Если не указано, то принимается NULL. В этом случае класс заданий не связывается ни с одним ресурсным планом базы данных и не имеет ограничений на использование ресурсов.
service	VARCHAR2	Имя сервиса (если в базе данных определены имена сервисов), которому должен принадлежать класс заданий.
logging_level	PLS_INTEGER	Объем журнальных данных, порождаемых заданиями, принадлежащими данному классу. Возможные значения: DBMS_SCHEDULER.LOGGING_OFF (запись в журнал не ведется), DBMS_SCHEDULER.LOGGING_RUNS (запись в журнал происходит только при выполнении заданий) и DBMS_SCHEDULER.LOGGING_FULL (помимо записей о выполнении

Имя параметра	Тип данных	Описание
		в журнал записываются все действия с заданиями, такие как ALTER, DROP).
log_history	PLS_INTEGER	Длительность хранения журнальных записей в днях. По прошествии этого срока записи удаляются.
comments	VARCHAR2	Комментарии к классу заданий.

STOP_JOB

Процедура останавливает выполнение задания.

Имя параметра	Тип данных	Описание
job_name	VARCHAR2	Имя останавливаемого задания. Можно указать разделенный запятыми список заданий (например, JOB1,JOB2). Либо можно указать имя класса заданий. В этом случае будут остановлены все задания данного класса.
force	BOOLEAN	Если задание в данный момент выполняется, то процедура пытается остановить его с помощью прерывания. Однако задание может не остановиться сразу, и команда останова вынуждена будет ждать. Если этот параметр установлен в значение TRUE, то процедура прервет порожденные заданием процессы. По умолчанию – FALSE.

RUN_JOB

Процедура инициирует немедленное выполнение задания, даже если это не предусмотрено расписанием.

Имя параметра	Тип данных	Описание
job_name	VARCHAR2	Имя задания.
use_current_session	BOOLEAN	При значении TRUE задание выполняется в текущем сеансе и сообщения немедленно выводятся на экран пользователю. Это очень полезно при отладке задания. При значении FALSE задание выполняется в фоновом режиме в другом сеансе, как и должно быть в нормальных условиях. По умолчанию – TRUE.

COPY_JOB

Процедура создает новое задание, используя определение уже существующего. Это очень полезно в тех случаях, когда надо быстро создать задание «такое же», как уже имеющееся, но с другим именем.

Имя параметра	Тип данных	Описание
old_job	VARCHAR2	Имя задания, атрибуты которого должны быть скопированы.
new_job	VARCHAR2	Имя нового задания, создаваемого на основе старого. Новое задание создается отключенным, следует явно включить его.

DISABLE

Процедура используется для отключения компонентов системы заданий. Она чрезвычайно полезна в тех случаях, когда необходимо временно остановить запланированное выполнение. Предположим, например, что у вас есть задание, вычисляющее и начисляющее пени, и в нем только что обнаружили ошибку. До тех пор пока ошибка не будет исправлена, задание apply_interest не должно выполняться. При отключении сохраняется вся информация о задании, оно просто не выполняется. Впоследствии можно включить его.

Данная процедура применяется для отмены всех типов составляющих системы планирования: заданий, классов заданий, расписаний, программ, окон и групп окон. Выбор объекта определяет поведение процедуры.

Имя параметра	Тип данных	Описание
name	VARCHAR2	Имя объекта (если речь идет о приложении, то перед именем указывается владелец объекта), который должен быть отключен. Можно указать несколько объектов, введя их имена в виде списка значений, разделенных запятыми.
force	BOOLEAN	Параметр ведет себя по-разному в зависимости от полученного имени. Как правило, значение TRUE вызывает отключение зависимых объектов, в то время как значение FALSE (умолчание) не вызывает отключения зависимых объектов.

ENABLE

Процедура дополняет собой процедуру DISABLE и используется для включения отключенных объектов. Как и ее родственница, процедура действует для всех типов составляющих системы планирования: заданий, классов заданий, расписаний, программ, окон и групп окон.

Имя параметра	Тип данных	Описание
name	VARCHAR2	Имя объекта (если речь идет о приложении, то перед именем указывается владелец объекта), который должен быть включен. Можно указать несколько объектов, введя их имена в виде списка значений, разделенных запятыми.

DROP_JOB

Процедура удаляет задание, в котором больше нет необходимости.

Имя параметра	Тип данных	Описание
job_name	VARCHAR2	Имя удаляемого задания. Можно указать несколько заданий, введя их имена в виде списка значений, разделенных запятыми (например, «JOB1,JOB2» и т. д.). Кроме того, можно указать имя класса заданий. В этом случае будут удалены все задания, входящие в данный класс. Обратите внимание, что удалить класс заданий этой процедурой невозможно. Для этого существует процедура DROP_JOB_CLASS.
force	BOOLEAN	Задание, выполняющееся в текущий момент, не может быть удалено. Значение TRUE этого параметра означает, что процедура будет пытаться остановить задание, прежде чем удалять его. Значение по умолчанию – FALSE, которое означает, что при вызове DROP_JOB будет сгенерировано сообщение об ошибке.

DROP_JOB_CLASS

Процедура удаляет класс заданий, в котором больше нет необходимости.

Имя параметра	Тип данных	Описание
job_class_name	VARCHAR2	Имя удаляемого класса заданий. Можно указать несколько классов, введя их имена в виде списка значений, разделенных запятыми (например, «JOB_CLASS1,JOB_CLASS2» и т. д.).
force	BOOLEAN	Если существует какое-то задание, которому присвоен данный класс, то такой класс удалить нельзя. Значение параметра TRUE означает, что задания будут помечены как отключенные, и удаление класса заданий будет разрешено. Значение по умолчанию – FALSE. Если в текущий момент работает здание, принадлежащее классу job_class, класс не удаляется.

CREATE_SCHEDULE

Процедура создает именованное расписание, которое может быть использовано при создании задания. Благодаря этому вы можете использовать вместо строки календарных значений именованное расписание.

Имя параметра	Тип данных	Описание
schedule_name	VARCHAR2	Имя расписания. Должно быть уникальным в рамках базы данных.
start_date	TIMESTAMP WITH TIMEZONE	Дата, с которой начинает действовать расписание. Значение по умолчанию – NULL (начать действовать незамедлительно).
repeat_interval	VARCHAR2	Календарная строка, определяющая, как часто должно повторяться расписание (например, FREQ=DAILY; BYHOUR=3). Полный перечень значений приведен в главе 8.
end_date	TIMESTAMP WITH TIMEZONE	Дата завершения работы расписания. Значение по умолчанию – NULL (никогда не заканчиваться).
comments	VARCHAR2	Комментарии для расписания.

DROP_SCHEDULE

Процедура удаляет именованное расписание.

Имя параметра	Тип данных	Описание
schedule_name	VARCHAR2	Имя удаляемого расписания.
force	BOOLEAN	Если на расписание ссылается задание или окно, то такое расписание удалить нельзя. Если параметр установлен в значение TRUE, то задание или окно будут отключены, и расписание будет удалено. Значение по умолчанию – FALSE.

CREATE_WINDOW

Эта программа создает именованное окно, которое может использоваться в качестве расписания для задания. Программа реализована в виде двух перегруженных процедур.

Процедура с именованным расписанием (версия 1)

Создает именованное окно. Расписание окна (то есть сведения о том, как часто окно должно повторяться) задается в виде именованного расписания, которое должно было быть определено ранее.

Имя параметра	Тип данных	Описание
window_name	VARCHAR2	Имя окна. Должно быть уникальным в рамках базы данных.
resource_plan	VARCHAR2	План диспетчера ресурсов, сопоставленный данному окну. Вы обязаны указать данный параметр и предоставить действительный ресурсный план.

Имя параметра	Тип данных	Описание
schedule_name	VARCHAR2	Именованное расписание, по которому будет действовать окно.
duration	INTERVAL DAY TO SECOND	Продолжительность пребывания окна в открытом состоянии.
window_priority	VARCHAR2	Приоритет окна. Возможные значения: LOW и HIGH; значение по умолчанию – LOW. Этот параметр важен только при перекрывании двух окон. Приоритет определяет, какое окно будет закрыто и уступит место другому.
comments	VARCHAR2	Комментарии для окна.

Процедура с встроенным расписанием (версия 2)

Идентична версии 1, только расписание задается в виде встроенной календарной строки.

Имя параметра	Тип данных	Описание
window_name	VARCHAR2	Имя окна. Должно быть уникальным в рамках базы данных.
resource_plan	VARCHAR2	План диспетчера ресурсов, сопоставленный данному окну. Вы обязаны указать данный параметр и предоставить действительный ресурсный план.
start_date	TIMESTAMP WITH TIMEZONE	Дата и время запуска первого открытия окна. Значение NULL (умолчание) означает, что окно будет открыто незамедлительно.
repeat_interval	VARCHAR2	Календарная строка, которая определяет, когда окно должно быть открыто повторно (например, FREQ=DAILY BYHOUR=3). Полный перечень значений приведен в главе 8. По умолчанию (NULL) задание никогда не будет повторяться, то есть будет выполнено единожды, в момент start_date.
end_date	TIMESTAMP WITH TIMEZONE	Если параметр задан, это означает, что по достижении его значения окно должно быть отключено. Значение по умолчанию – NULL, которое означает, что окно будет открываться бесконечно.
duration	INTERVAL DAY TO SECOND	Продолжительность пребывания окна в открытом состоянии.
window_priority	VARCHAR2	Приоритет окна. Возможные значения: LOW и HIGH; значение по умолчанию – LOW. Этот параметр важен только при перекрывании двух окон. Приоритет определяет, какое окно будет закрыто и уступит место другому.
comments	VARCHAR2	Комментарии для окна.

CREATE_WINDOW_GROUP

Процедура создает именованную группу окон. Группа окон может использоваться как расписание для выполнения заданий.

Имя параметра	Тип данных	Описание
group_name	VARCHAR2	Имя группы окон. Должно быть уникальным в рамках базы данных.
window_list	VARCHAR2	Разделенный запятыми список окон, входящих в группу (например, WIN1,WIN2). Можно создать группу, в которую пока не будут входить никакие окна: для этого установите данный параметр в значение NULL.
comments	VARCHAR2	Комментарии для группы окон.

ADD_WINDOW_GROUP_MEMBER

При создании группы окон можно определить группу, не включающую в себя ни одного элемента. Впоследствии вы можете добавить в такую группу окна, используя данную процедуру. Также можно добавлять окна в группы, уже включающие в себя какие-то элементы.

Имя параметра	Тип данных	Описание
group_name	VARCHAR2	Имя группы окон, которая должна быть создана ранее.
window_list	VARCHAR2	Разделенный запятыми список окон, добавляемых в группу (например, WIN1,WIN2).

DROP_WINDOW

Процедура удаляет окно.

Имя параметра	Тип данных	Описание
group_name	VARCHAR2	Имя удаляемого окна.
force	BOOLEAN	Если окно используется заданием в качестве расписания, то удалить такое окно невозможно. При установке параметра в значение TRUE задание будет отключено, а окно – удалено. Значение по умолчанию – FALSE.

OPEN_WINDOW

Процедура открывает окно «вручную». Окно, для которого еще не наступил предусмотренный расписанием момент открытия, считается закрытым. С помощью процедуры можно открыть закрытое окно вручную. Тем самым будут запущены все задания, связанные с данным окном.

Имя параметра	Тип данных	Описание
window_name	VARCHAR2	Имя открываемого окна.
duration	INTERVAL DAY TO SECOND	Продолжительность действия данного окна.
force	BOOLEAN	Если окно уже открыто, вы не можете заново открыть его, и процедура возвращает ошибку. Если параметр установлен в значение TRUE, то открытие окна не приведет к ошибке, а приведет лишь к изменению продолжительности действия окна на значение, указанное в предыдущем параметре (начиная с текущего момента).

CLOSE_WINDOW

Процедура позволяет вручную закрыть окно до истечения срока его действия.

Имя параметра	Тип данных	Описание
window_name	VARCHAR2	Имя закрываемого окна. Окно должно быть открыто на текущий момент; в противном случае будет сгенерирована ошибка.

Представления словаря данных для планировщика

В этом разделе будут рассмотрены столбцы представлений словаря данных, относящиеся к планировщику заданий.

DBA_SCHEDULER_JOBS

Представление отображает данные о заданиях, которые определены в базе данных.

Имя столбца	Описание
OWNER	Владелец задания.
JOB_NAME	Имя задания.
CLIENT_ID	Если при создании задания пользователь указал идентификатор клиента для сеанса, он будет записан в данный столбец. Для задания идентификатора клиента можно вызвать функцию DBMS_SESSION.SET_IDENTIFIER.
GLOBAL_UID	Если пользователь является глобальным (или корпоративным), то в данный столбец записывается идентификатор глобального пользователя.
JOB_TYPE	Тип задания; допустимы следующие значения: EXECUTABLE, PLSQL_BLOCK и STORED_PROCEDURE.
JOB_ACTION	Что делает задание. Для PL/SQL-блока отображается весь код. Для исполняемого файла и хранимой процедуры отображается название.

Имя столбца	Описание
START_DATE	Время начала выполнения задания как значение типа TIMESTAMP.
REPEAT_INTERVAL	Календарная строка, задающая расписание выполнения задания (например, FREQ=DAILY; BYHOUR=2). (См. также раздел «Календарные строки» в главе 8.)
ENABLED	Указывает, включено ли задание (TRUE или FALSE).
STATE	Текущий статус задания (например, SCHEDULED, RUNNING, SUCCEEDED, FAILED).
RUN_COUNT	Количество выполненных запусков задания.
FAILURE_COUNT	Количество неудачных завершений задания.
RETRY_COUNT	В случае неудачного выполнения задания предпринимается повторная попытка выполнения. В данном столбце отображается количество повторных попыток.
LAST_START_DATE	Временная метка последнего запуска задания.
LAST_RUN_DURATION	Продолжительность последнего выполнения задания.
NEXT_RUN_DATE	Следующая запланированная дата выполнения задания.
SYSTEM	Указывает, является ли задание системным заданием (TRUE или FALSE).
COMMENTS	Введенные ранее комментарии.

В Oracle 10*g* Release 2 вводится новая функциональность – *задания, управляемые событиями* (*event-based jobs*) (не описанная в данной книге), которая позволяет выполнять некоторые задания в соответствии не со временем, а с наступлением определенных событий. Для поддержки этой функциональности Oracle 10*g* Release 2 добавляет в представление DBA_SCHEDULER_JOBS следующие новые столбцы.

Имя столбца	Описание
JOB_SUBNAME	Дополнительное имя задания (имя шага).
SCHEDULE_TYPE	Тип расписания. Допустимы следующие значения: ONCE, CALENDAR и EVENT.
EVENT_QUEUE_OWNER	Имя владельца очереди событий.
EVENT_QUEUE_NAME	Имя очереди событий.
EVENT_QUEUE_AGENT	Имя агента очереди событий.
EVENT_CONDITION	Условие, вызывающее наступление события.
EVENT_RULE	Правило, управляющее запуском события и последующим выполнением задания.
RAISE_EVENTS	После завершения задания другое событие (события) может запустить другое задание. Такое событие (события) будет зарегистрировано в этом столбце.

DBA_SCHEDULER_WINDOWS

Представление отображает сведения об окнах планировщика, которые определены в базе данных.

Имя столбца	Описание
WINDOW_NAME	Имя окна.
RESOURCE_PLAN	Имя ресурсного плана, связанного с окном.
SCHEDULE_OWNER	Если окно имеет именованное расписание, то имя владельца такого расписания выводится в данном столбце.
SCHEDULE_NAME	Имя расписания, если оно существует.
START_DATE	Временная метка открытия окна. Действует только в том случае, когда для окна задана встроенная календарная строка, а не именованное расписание. Выводится как значение типа TIMESTAMP(6) WITH TIMEZONE.
REPEAT_INTERVAL	Если для окна задана встроенная календарная строка, а не именованное расписание, то эта календарная строка выводится в данном столбце (см. раздел «Календарные строки» в главе 8).
END_DATE	Если для окна задана встроенная календарная строка, а не именованное расписание, то в этом столбце выводится метка окончательного закрытия окна. Выводится как значение типа TIMESTAMP(6) WITH TIMEZONE.
DURATION	Продолжительность периода, в течение которого окно будет открытым. Выводится как значение типа DURATION.
WINDOW_PRIORITY	При наложении двух окон окно с более высоким приоритетом будет открыто, а второе – закрыто. Приоритеты отображаются как значения HIGH и LOW.
NEXT_START_DATE	Временная метка следующего запланированного открытия окна. Выводится как значение типа TIMESTAMP(6) WITH TIMEZONE.
LAST_START_DATE	Временная метка последнего открытия окна. Выводится как значение типа TIMESTAMP(6) WITH TIMEZONE.
ENABLED	Указывает, включено ли окно (TRUE/FALSE).
ACTIVE	Указывает, открыто ли окно в настоящий момент (TRUE/FALSE).
COMMENTS	Комментарии для окна.

В Oracle 10*g* Release 2 в представление DBA_SCHEDULER_WINDOWS добавляется несколько новых столбцов, которые очень полезны в случае, если окно открывается вручную.

Имя столбца	Описание
SCHEDULE_TYPE	Тип расписания. Возможны следующие значения: ONCE, CALENDAR и EVENT.

Имя столбца	Описание
MANUAL_OPEN_TIME	Если окно было открыто вручную, то время открытия отображается в данном столбце в виде значения типа TIMESTAMP.
MANUAL_DURATION	Если окно было открыто вручную, то его продолжительность отображается в данном столбце в виде значения типа INTERVAL.

DBA_SCHEDULER_SCHEDULES

Представление отображает сведения об именованных расписаниях, которые определены в базе данных.

Имя столбца	Описание
OWNER	Владелец расписания.
SCHEDULE_NAME	Имя расписания.
START_DATE	Дата и время предполагаемого начала действия расписания.
REPEAT_INTERVAL	Календарная строка, которая указывает, как часто будет повторяться расписание. Использует синтаксис календарной строки (например, FREQ=DAILY; BYMONTH=2). Подробное описание синтаксиса приведено в разделе «Календарные строки» главы 8.
END_DATE	Дата завершения действия расписания.
COMMENTS	Комментарии для расписания.

В Oracle 10*g* Release 2 в представление DBA_SCHEDULER_SCHEDULES добавляется несколько новых столбцов для поддержки механизма заданий, управляемых событиями.

Имя столбца	Описание
SCHEDULE_TYPE	Тип расписания. Возможны следующие значения: ONCE, CALENDAR и EVENT.
EVENT_QUEUE_OWNER	Имя владельца очереди событий.
EVENT_QUEUE_NAME	Имя очереди событий.
EVENT_QUEUE_AGENT	Имя агента очереди событий.
EVENT_CONDITION	Условие, вызывающее наступление события.

DBA_SCHEDULER_PROGRAMS

Представление отображает сведения об именованных программах из базы данных.

Имя столбца	Описание
OWNER	Владелец программы.
PROGRAM_NAME	Имя программы.

Имя столбца	Описание
PROGRAM_TYPE	Тип программы. Допустимы следующие значения: EXECUTABLE, PLSQL_BLOCK и STORED_PROCEDURE.
PROGRAM_ACTION	Что делает программа. Для PL/SQL-блока отображается весь код. Для хранимой процедуры отображается название. Для исполняемого файла – полный путь.
NUMBER_OF_ARGUMENTS	Количество аргументов программы (если они есть). Если аргументов нет, выводится 0.
ENABLED	Указывает, включена ли программа (TRUE/FALSE).
COMMENTS	Комментарии для программы.

DBA_SCHEDULER_JOB_CLASSES

Представление отображает сведения о классах заданий из базы данных.

Имя столбца	Описание
JOB_CLASS_NAME	Имя класса заданий.
RESOURCE_CONSUMER_GROUP	Имя группы потребителей ресурсов, которая управляет выделением ресурсов для данного класса заданий.
SERVICE	Имя сервиса, который должен использоваться данным классом заданий.
LOGGING_LEVEL	Объем журнальных данных, порождаемых заданиями, принадлежащими данному классу. Возможные значения: OFF (запись в журнал не ведется), RUNS (запись в журнал происходит только при выполнении заданий) и FULL (в журнал записываются все действия с заданиями (например, ALTER, DROP)).
LOG_HISTORY	Количество дней, в течение которых хранится журнал.
COMMENTS	Комментарии для класса заданий.

DBA_SCHEDULER_WINDOW_GROUPS

Представление отображает сведения о группах окон, которые определены в базе данных.

Имя столбца	Описание
WINDOW_GROUP_NAME	Имя группы окон.
ENABLED	Указывает, включена ли группа окон (TRUE/FALSE).
NUMBER_OF_WINDOWS	Количество окон, включенных в данную группу. Имена окон, принадлежащих группе, доступны в представлении DBA_SCHEDULER_WINGROUP_MEMBERS.
COMMENTS	Комментарии для группы окон.

DBA_SCHEDULER_WINGROUP_MEMBERS

Представление отображает сведения о группах окон и об окнах, принадлежащих таким группам.

Имя столбца	Описание
WINDOW_GROUP_NAME	Имя группы окон.
WINDOW_NAME	Имя окна.

DBA_SCHEDULER_JOB_LOG

Подробное описание этого представления приведено в разделе «Управление журналированием» главы 8.

DBA_SCHEDULER_JOB_RUN_DETAILS

Подробное описание этого представления приведено в разделе «Управление журналированием» главы 8.

DBA_SCHEDULER_RUNNING_JOBS

Представление отображает сведения обо всех заданиях, выполняющихся в текущий момент в базе данных.

Имя столбца	Описание
OWNER	Владелец задания.
JOB_NAME	Имя задания.
SESSION_ID	Идентификатор сеанса (из V$SESSION) для выполняемого задания.
SLAVE_PROCESS_ID	Идентификатор подчиненного процесса.
RUNNING_INSTANCE	Номер экземпляра базы данных, на котором работает данное задание. Этот параметр имеет смысл только для кластерной базы данных.
RESOURCE_CONSUMER_GROUP	Имя группы потребителей ресурсов, с которой связано данное задание. Управляет выделением ресурсов (ЦПУ, сервер параллельных запросов и т. д.) заданию.
ELAPSED_TIME	Время, прошедшее с момента запуска данного задания. Выводится как значение типа INTERVAL.
CPU_USED	Количество циклов ЦПУ (в единицах времени), использованных данным заданием. Выводится как значение типа INTERVAL.

В Oracle 10*g* Release 2 в представление DBA_SCHEDULER_RUNNING_JOBS добавляется несколько новых столбцов.

Имя столбца	Описание
JOB_SUBNAME	Дополнительное имя задания (имя шага).
SLAVE_OS_PROCESS_ID	Идентификатор процесса операционной системы для подчиненного процесса.

Алфавитный указатель

Специальные символы

A

B

C

D

E

F

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

A

Б

Ж

З

И

К

Л

М

Н

О

П

P

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Я