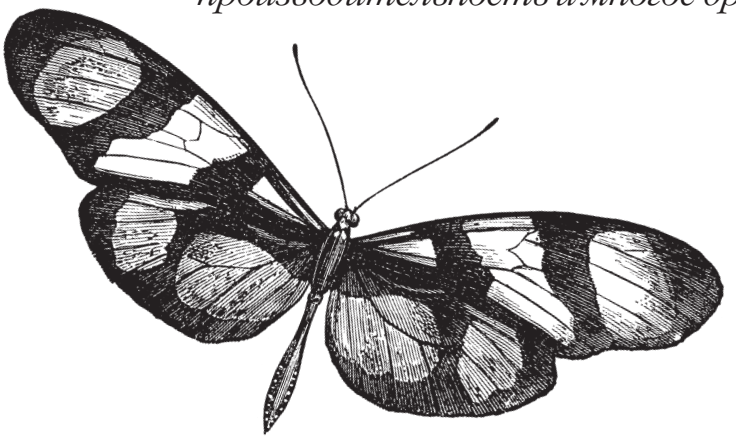
Безопасность, планирование, производительность и многое другое

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.

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.

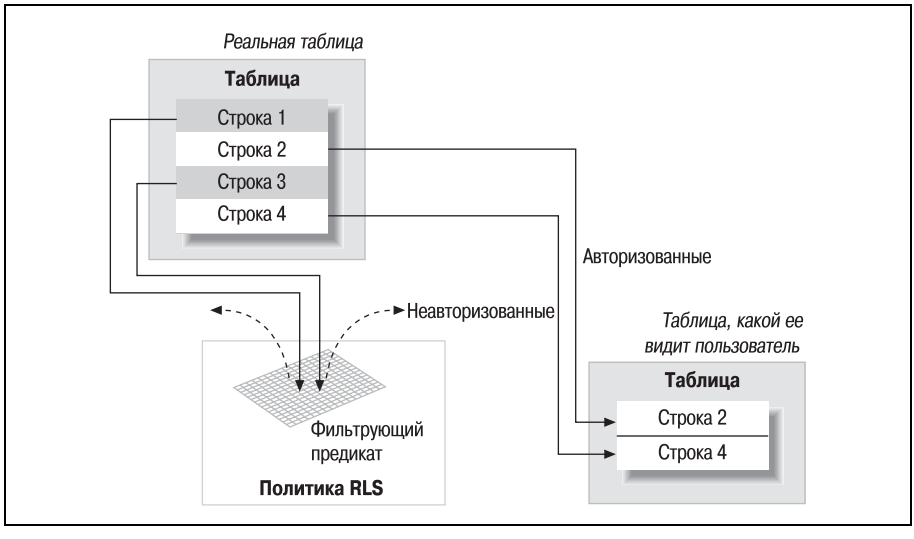


Рис. 5.1. *Работа фильтра, обеспечивающего безопасность на уровне строк*

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.

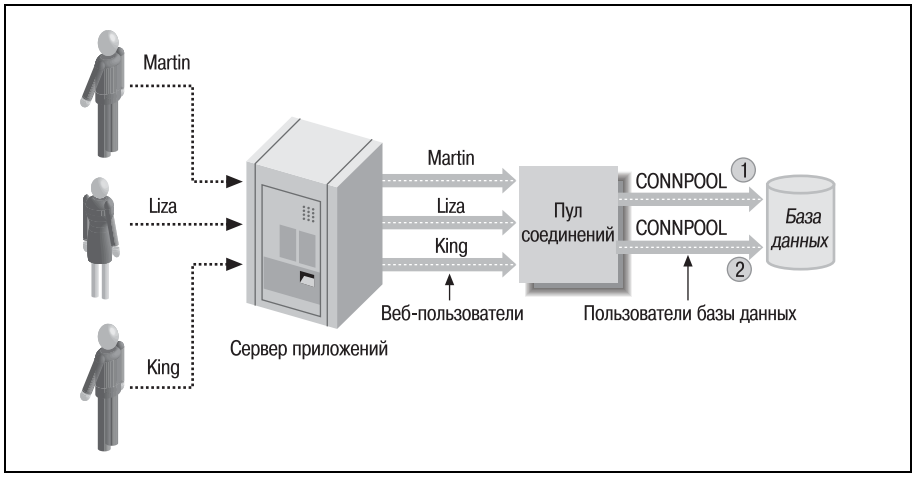


Рис. 5.2. *Пользователи приложений и RLS*

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.