

# Секреты Oracle SQL

*Санжей Мишира и Алан Бьюли*



---

*Санкт-Петербург — Москва*  
*2003*

# Оглавление

Предисловие .....	11
<b>1. Введение в SQL .....</b>	<b>19</b>
Что такое SQL? .....	19
Краткая история SQL .....	21
Простая база данных .....	22
Операторы DML .....	24
<b>2. Инструкция WHERE .....</b>	<b>33</b>
Жизнь без WHERE .....	33
На помощь приходит WHERE .....	34
Вычисление инструкции WHERE .....	36
Условия и выражения .....	38
Куда идем дальше? .....	44
<b>3. Объединения .....</b>	<b>46</b>
Внутренние объединения .....	47
Внешние объединения .....	50
Самообъединения .....	59
Объединения и подзапросы .....	64
Операторы DML и представление объединения .....	64
Синтаксис объединения стандарта ANSI в Oracle9i .....	71
<b>4. Групповые операции .....</b>	<b>78</b>
Обобщающие функции .....	78
Инструкция GROUP BY .....	82
Инструкция HAVING .....	88
<b>5. Подзапросы .....</b>	<b>91</b>
Что такое подзапрос? .....	91
Несвязанные подзапросы .....	92
Связанные подзапросы .....	99

Встроенные представления .....	101
Изучаем пример подзапроса: N лучших работников .....	115
<b>6. Обработка дат и времени</b> .....	121
Внутренний формат хранения даты .....	122
Вставка дат в БД и извлечение дат из БД .....	122
Работа с датами .....	137
Новые возможности Oracle9i по обработке даты и времени .....	152
Литералы INTERVAL .....	160
<b>7. Операции над множествами</b> .....	173
Операторы работы с множествами .....	174
Использование операций над множествами для сравнения двух таблиц .....	178
Использование NULL в составных запросах .....	181
Правила и ограничения, налагаемые на операции над множествами .....	183
<b>8. Иерархические запросы</b> .....	187
Представление иерархической информации .....	187
Простые операции над иерархическими данными .....	190
Расширения Oracle SQL .....	193
Сложные иерархические операции .....	198
Ограничения, налагаемые на иерархические запросы .....	205
<b>9. DECODE и CASE</b> .....	207
DECODE, NVL и NVL2 .....	207
История CASE .....	211
Примеры использования DECODE и CASE .....	214
<b>10. Разделы, объекты и коллекции</b> .....	226
Разделение таблиц .....	226
Объекты и коллекции .....	237
<b>11. PL/SQL</b> .....	249
Что такое PL/SQL? .....	249
Процедуры, функции и пакеты .....	250
Вызов хранимых функций из запросов .....	252
Ограничения на вызов PL/SQL из SQL .....	257
Хранимые функции в операторах DML .....	261
SQL внутри PL/SQL .....	263

<b>12. Сложные групповые операции</b> .....	266
ROLLUP .....	266
CUBE .....	276
Функция GROUPING .....	283
GROUPING SETS .....	288
Возможности группировки в Oracle9i .....	289
Функции GROUPING_ID и GROUP_ID .....	299
<b>13. Аналитический SQL</b> .....	307
Обзор аналитического SQL .....	307
Ранжирующие функции .....	313
Оконные функции .....	327
Функции для создания отчетов .....	333
Резюме .....	338
<b>14. Советы умудренных опытом</b> .....	339
Когда и какие конструкции использовать? .....	339
Избегайте ненужного разбора операторов .....	345
Применяйте полностью определенный SQL для систем поддержки принятия решений .....	350
Алфавитный указатель .....	352



# Предисловие

SQL (Structured Query Language, язык структурированных запросов) – это язык, применяемый для доступа к реляционной базе данных (БД). SQL состоит из набора операторов, позволяющих сохранять данные в БД и извлекать их из БД. Популярность языка неуклонно росла с того самого момента, когда на свет появилась первая реляционная база данных. Предлагались и другие языки запросов, но на данный момент SQL признан стандартным языком почти для всех реализаций реляционных баз данных, включая Oracle.

SQL отличается от других языков программирования тем, что он не процедурный. В отличие от программ на других языках, в которых необходимо задавать последовательность действий для выполнения, программа на SQL (более точно называемая SQL-оператором) всего лишь описывает желаемый результат. Система управления базами данных сама определяет, каким образом следует обработать данные для получения желаемого результата. Непроцедурная природа SQL облегчает доступ к данным из прикладных программ.

Если вы пользуетесь базой данных Oracle, то SQL – это интерфейс, который вы применяете для доступа к данным, хранящимся в базе. SQL позволяет создавать структуры базы данных, такие как таблицы (для хранения данных), представления и индексы. SQL позволяет заносить данные в БД и извлекать хранимые данные в нужном формате (например, отсортированными). Наконец, SQL позволяет модифицировать, удалять и всячески манипулировать хранимыми данными. SQL – это ключ ко всем вашим действиям с базой данных. Совершенное владение языком SQL является одним из наиболее важных требований к разработчику или администратору баз данных.

## Зачем мы написали эту книгу

Наши собственные впечатления от изучения и применения базы данных Oracle и реализации SQL от Oracle побудили нас написать эту книгу. Документация Oracle по SQL состоит из справочного руководства, которое не углубляется в тонкости практического применения различ-

ных особенностей SQL, поддерживаемых Oracle, и не содержит сложных примеров из реальной жизни.

Обратившись за помощью в магазин компьютерной литературы, мы обнаружили, что в действительности существуют лишь два типа книг по SQL. Большинство из них справочного типа; они описывают функции и синтаксис языка, но ничего не говорят о том, как применить эти знания к реальным задачам. Другой тип книг, весьма малочисленный, обсуждает применение SQL в сухом теоретическом стиле, не используя какую-либо реализацию конкретного производителя. А поскольку каждый производитель баз данных реализует собственный вариант SQL, мы считаем книги по «стандартному» SQL не очень-то полезными.

Мы решили написать практическое руководство, сконцентрировавшись непосредственно на Oracle-версии SQL. Oracle лидирует на рынке СУБД, к тому же, это та база данных, с помощью которой мы приобрели свои навыки работы с SQL. В данной книге мы не только представляем наиболее важные и полезные свойства Oracle SQL, но и описываем пути их применения для решения конкретных задач.

## Цели этой книги

Единственная и важнейшая цель этой книги – помочь вам максимально использовать всю мощь Oracle SQL. Вы научитесь:

- Понимать особенности и возможности языка SQL в реализации Oracle.
- Использовать такие возможности SQL, как внешние объединения, связанные подзапросы, иерархические запросы, группирующие операции, аналитические запросы и т. д.
- Использовать инструкции DECODE и CASE для внесения условной логики в SQL-запросы.
- Строить SQL-запросы, работающие с разделами, объектами и коллекциями, такими как вложенные таблицы и массивы переменной длины.
- Использовать новые возможности SQL, представленные в Oracle9i, такие как новые особенности работы с датой и временем, ANSI-совместимые объединения и новые группирующие и аналитические функции.
- Использовать лучшие приемы для написания эффективных и удобных для сопровождения SQL-запросов.

## Для кого наша книга?

Эта книга предназначена для разработчиков и администраторов баз данных Oracle. Новичок ли вы в мире баз данных или опытный специалист – если вы используете SQL для доступа к базе данных Oracle, эта



книга для вас. Вне зависимости от того, используете ли вы простые запросы для доступа к данным или же встраиваете их в PL/SQL- или Java-программы, SQL является основой для всех задач доступа к данным в вашем приложении. Овладев мощными и гибкими средствами SQL, вы повысите производительность своего труда, будете успевать сделать больше за меньшее время и при этом будете уверены в правильности написанных вами SQL-запросов.

## Платформа и версия

В этой книге мы использовали Oracle8i (версии 8.1.6 и 8.1.7) и Oracle9i (версия 9.0.1). Мы описали многие новые важные возможности Oracle9i SQL, включая синтаксис объединений, соответствующий стандарту ANSI, новые типы для времени и даты, различные аналитические функции. Большинство же понятий, правил синтаксиса и примеров относится и к более ранним версиям Oracle. Новые возможности Oracle9i мы отмечаем специально.

## Структура этой книги

Книга состоит из 14 глав:

- Глава 1 «Введение в SQL» знакомит читателя с языком SQL и дает краткое представление о его истории. Эта глава предназначена в основном для тех читателей, которые имеют небольшой опыт или вообще не имеют никакого опыта работы с SQL. В ней вы найдете простые примеры основных операторов SQL (SELECT, INSERT, UPDATE и DELETE) и стандартных возможностей SQL.
- В главе 2 «Инструкция WHERE» описываются способы фильтрации данных в операторах SQL. Вы научитесь ограничивать результаты запроса теми строками, которые хотите видеть, и распространять результаты выполнения оператора на те строки, которые хотите изменить.
- В главе 3 «Объединения» описываются конструкции, используемые для доступа к данным из нескольких связанных таблиц. В этой главе обсуждаются такие важные понятия, как внутренние и внешние объединения. Также описывается новый, соответствующий стандарту ANSI синтаксис объединений, представленный в Oracle9i.
- В главе 4 «Групповые операции» показывается, как формировать суммарную информацию, такую как итоговую и подытоговую суммы ваших данных. Вы узнаете, как определять группы строк и как применять различные обобщающие функции для обработки данных из этих групп.
- В главе 5 «Подзапросы» рассказывается, как использовать связанные и несвязанные подзапросы и встроенные представления для ре-

шения сложных проблем, которые без использования этих средств потребовали бы процедурного кода и нескольких запросов к БД.

- Глава 6 «Обработка дат и времени» рассказывает о том, как обращаться с датой и временем в базе данных Oracle. Вы познакомитесь с различными приемами, используемыми при запрашивании данных о времени. Также вы узнаете о многих новых типах данных для представления даты и времени, введенных в Oracle9i.
- В главе 7 «Операции над множествами» показывается, как использовать инструкции UNION, INTERSECT и MINUS для комбинирования результатов двух или более независимых запросов.
- В главе 8 «Иерархические запросы» обсуждается, как хранить и извлекать иерархическую информацию (такую как структура организации) из реляционной таблицы. Oracle обладает рядом возможностей, облегчающих работу с иерархическими данными.
- Глава 9 «DECODE и CASE» рассказывает о двух очень мощных, но в то же время простых возможностях Oracle SQL, позволяющих имитировать условную логику (без них SQL был бы просто декларативным языком). CASE, стандартная конструкция ANSI, была впервые представлена в Oracle8i и улучшена в Oracle9i.
- В главе 10 «Разделы, объекты и коллекции» обсуждаются вопросы, связанные с доступом к разделам и коллекциям посредством SQL. Вы научитесь строить операторы SQL, которые работают с отдельными разделами и подразделами. Также вы узнаете, как запрашивать объектные данные, вложенные таблицы и массивы переменной длины.
- Глава 11 «PL/SQL» посвящена интеграции SQL и PL/SQL. Вы узнаете, как вызывать хранимые процедуры и функции PL/SQL из SQL-запросов и как писать эффективные операторы SQL в PL/SQL-программах.
- В главе 12 «Сложные групповые операции» рассматриваются сложные групповые операции, использующиеся в основном в системах принятия решений. Мы покажем, как использовать возможности Oracle, такие как ROLLUP, CUBE и GROUPING SETS, для успешного формирования различных уровней суммарной информации, необходимой в системах принятия решений. Кроме того, обсуждаются новые групповые особенности Oracle9i, которые делают возможными составные и сцепленные группировки, а также новые функции GROUP\_ID и GROUPING\_ID.
- Глава 13 «Аналитический SQL» знакомит вас с аналитическими запросами и новыми аналитическими функциями. Вы узнаете, как использовать ранжирующие, оконные функции и функции составления отчетов, чтобы сформировать информацию для системы принятия решений. Эта глава также охватывает новые аналитические возможности, представленные в Oracle9i.



- Глава 14 «Советы умудренных опытом» рассказывает о правилах, которых следует придерживаться, чтобы писать эффективные и удобные в обслуживании запросы. Вы узнаете, какие конструкции SQL наиболее эффективны в конкретной ситуации. Например, мы объясняем, когда лучше использовать WHERE вместо HAVING для ограничения результатов запроса. Также обсуждается влияние на производительность использования связанных переменных по сравнению с полностью определенным SQL.

## Соглашения, используемые в этой книге

В оформлении книги приняты следующие соглашения:

### *Курсив*

Используется для имен файлов и каталогов, названий таблиц и полей, а также для URL. Кроме того, курсивом выделяются специальные термины при первом их появлении в тексте.

### Моноширинный шрифт

Применяется в примерах, а также для отображения содержимого файлов и выводимых данных.

### *Моноширинный курсив*

Применяется в описании синтаксиса для данных, которые должны быть заменены пользователем на реальные значения.

### Моноширинный полужирный шрифт

Используется в примерах для информации, которую вводит пользователь. Также применяется для выделения участков кода, на которые следует обратить внимание.

### *Моноширинный полужирный курсив*

Применяется в коде примеров для выделения конструкций SQL или результатов, которые обсуждаются в тексте.

## ПРОПИСНЫЕ БУКВЫ

Прописными буквами в описании синтаксиса выделяются ключевые слова.

### строчные буквы

Строчными буквами в описании синтаксиса выделяются определяемые пользователем элементы, например переменные.

### [ ]

В описании синтаксиса в квадратные скобки заключены необязательные элементы.

### { }

В описании синтаксиса в фигурные скобки заключается набор элементов, из которых должен быть выбран один.

В описании синтаксиса вертикальная черта разделяет элементы списка, помещаемого в фигурные скобки, например: {TRUE|FALSE}.

...

В описании синтаксиса многоточие указывает на повторение элемента.

Особое внимание уделяйте замечаниям, выделенным в тексте следующим образом:



Это совет, предложение или примечание. Например, мы используем примечания, для того чтобы отметить новые возможности Oracle9i.



Это предупреждение. Например, подобным образом отмечаются инструкции SQL, неосторожное применение которых может привести к непредвиденным последствиям.

## Комментарии и вопросы

Мы протестировали и проверили информацию в этой книге настолько хорошо, насколько это было возможно, но если вы обнаружите, что некоторые свойства изменились или мы допустили ошибки, сообщите, пожалуйста, об этом по адресу:

O'Reilly & Associates  
1005 Gravenstein Highway North  
Sebastopol, CA 95472  
(800) 998-9938 (in the United States or Canada)  
(707) 829-0515 (international or local)  
(707) 829-0104 (FAX)

Также вы можете послать сообщение по электронной почте. При желании попасть в список рассылки или получить каталог, напишите по адресу:

*[info@oreilly.com](mailto:info@oreilly.com)*

Можно задать технические вопросы или прокомментировать книгу, написав по адресу:

*[bookquestions@oreilly.com](mailto:bookquestions@oreilly.com)*

На веб-сайте книги можно найти список опечаток (найденные ошибки и исправления доступны для публичного просмотра):

*<http://www.oreilly.com/catalog/mastorasql>*



Для получения дополнительной информации об этой и других книгах посетите сайт O'Reilly:

<http://www.oreilly.com>

## Благодарности

Мы в долгу перед множеством людей, внесших вклад в подготовку и издание этой книги. Мы глубоко обязаны Джонатану Геннику (Jonathan Gennic), редактору этой книги. Его видение книги, внимательное отношение к деталям и исключительные редакторские навыки – вот причины, по которым эта книга сегодня перед вами.

Искренняя благодарность нашим техническим рецензентам: Дайане Лоренц (Diana Lorentz), Джеффу Коксу (Jeff Cox), Стефану Андерту (Stephan Andert), Ричу Уайту (Rich White), Петеру Линсли (Peter Linsley) и Крису Ли (Chris Lee), которые посвятили массу своего бесценного времени чтению и комментированию черновика книги. Их вклад сделал ее более точной и ценной, облегчил ее чтение.

Эта книга не смогла бы появиться без огромной работы и поддержки высококвалифицированных сотрудников O'Reilly & Associates, включая дизайнеров обложки Элли Волькхаусен (Ellie Volckhausen) и Эмму Колби (Emma Colby), дизайнера книги Дэвида Футато (David Futato), Нила Уолса (Neil Walls), который конвертировал файлы, редактора и выпускающего редактора Коллина Гормана (Coleen Gorman), иллюстраторов Роба Романо (Rob Romano) и Джессамин Рид (Jessamyn Read), а также Шерил Авруч (Sheryl Avguch) и Анну Ширмер (Ann Schirmer), контролировавших качество, и Тома Динса (Tom Dinse), составителя индекса. Также спасибо Тиму О'Рейлли (Tim O'Reilly) за просмотр книги и ценный отзыв.

## От Санжея

Сердечная благодарность моему соавтору Алану за его великолепные технические навыки и совместную работу над книгой. Особая благодарность Джонатану Геннику не только за редактирование книги, но и за предоставленный мне удаленный доступ к его базе данных Oracle9i.

Мои приключения, связанные с Oracle, начались в проекте Tribology Workbench в Тата Стил, Джемшедпур (Индия). Искренняя благодарность моим коллегам по проекту Tribology Workbench за все эксперименты и исследования, сделанные в ходе изучения Oracle. Особая благодарность Сарошу Мунчержи (Sarosh Muncherji), заместителю руководителя группы, за то, что он пригласил меня в проект и погрузил в мир Oracle, поручив мне администрирование базы данных. С тех пор технология баз данных Oracle стала для меня образом жизни.

Искренняя благодарность моим коллегам в i2 Technologies за их поддержку.

Последняя в списке, но не последняя по значимости благодарность моей жене Сьюдипти за ее понимание и поддержку.

## От Алана

Я хотел бы поблагодарить моего соавтора Санжея и редактора Джонатана Генника за то, что они разделили мои взгляды на эту книгу, и за их технический и редакторский героизм. Я никогда бы не дошел до финиша без вашей помощи и поддержки.

Больше всего я хотел бы поблагодарить мою жену Нэнси за ее поддержку и терпение и моих дочерей, Мишель и Николь, за их любовь и понимание.



# 1

## Введение в SQL

Во вступительной главе мы поговорим об истоках языка SQL, обсудим его наиболее полезные возможности и определим простую базу данных, которая ляжет в основу большинства примеров книги.

### Что такое SQL?

SQL (Structured Query Language, язык структурированных запросов) – это специальный язык, используемый для определения данных, доступа к данным и их обработки. SQL относится к *непроцедурным (non-procedural)* языкам – он лишь описывает нужные компоненты (например, таблицы) и желаемые результаты, не указывая, как именно эти результаты должны быть получены. Каждая реализация SQL является надстройкой над *процессором базы данных (database engine)*, который интерпретирует операторы SQL и определяет порядок обращения к структурам БД для корректного и эффективного формирования желаемого результата.

Язык SQL состоит из двух специальных наборов команд. DDL (Data Definition Language, язык определения данных) – это подмножество SQL, используемое для определения и модификации различных структур данных, а DML (Data Manipulation Language, язык манипулирования данными) – это подмножество SQL, применяемое для получения и обработки данных, хранящихся в структурах, определенных ранее с помощью DDL. DDL состоит из большого количества команд, необходимых для создания таблиц, индексов, представлений и ограничений, а в DML входит всего четыре оператора:

#### *INSERT*

Добавляет данные в базу данных.

## UPDATE

Изменяет данные в базе данных.

## DELETE

Удаляет данные из базы данных.

## SELECT

Извлекает данные из базы данных.

Некоторым кажется, что применение DDL является прерогативой администраторов базы данных, а операторы DML должны писать разработчики, но эти два языка не так-то просто разделить. Сложно организовать эффективный доступ к данным и их обработку, не понимая, какие структуры доступны и как они связаны. Также сложно проектировать соответствующие структуры, не зная, как они будут обрабатываться. Сказав это, сосредоточимся на DML. DDL в книге будет встречаться лишь тогда, когда это необходимо для иллюстрации применения DML. Причины особого внимания к DML таковы:

- DDL хорошо описан во многих книгах по проектированию и администрированию баз данных, а также в справочниках по SQL.
- Проблемы производительности обычно бывают вызваны неэффективными операторами DML.
- Хотя операторов всего четыре, DML – настолько большая тема, что ее хватило бы на целую серию книг.<sup>1</sup>

Но почему вообще кого-то должен интересовать SQL? Зачем в наш век Интернета и многоуровневых архитектур заботиться о доступе к данным? На самом деле эффективное хранение и извлечение информации сейчас важно как никогда ранее:

- Все больше компаний предлагают свои услуги через Интернет. В часы пик они вынуждены обслуживать тысячи параллельных запросов, и задержки означают прямую потерю прибыли. Для таких систем каждый оператор SQL должен быть тщательно продуман, чтобы обеспечивать требуемую производительность при увеличении объема данных.
- Сегодня есть возможность хранить гораздо больше данных, чем пять лет назад. Один дисковый массив вмещает десятки терабайт данных, и уже не за горами хранение сотен терабайт. Программное обеспечение, применяемое для загрузки и анализа данных в этих средах, должно использовать весь потенциал SQL, чтобы обрабатывать неизменно увеличивающийся объем данных за постоянные (или сокращающиеся) промежутки времени.

---

<sup>1</sup> Каждый, кто работает с SQL в среде Oracle, должен вооружиться тремя книгами: справочником по языку SQL, таким как «Oracle SQL: The Essential Reference» (O'Reilly), руководством по оптимизации производительности, например «Oracle SQL Tuning Pocket Reference» (O'Reilly), и этой книгой, которая показывает, как наилучшим образом использовать и комбинировать разнообразные возможности Oracle SQL.



Надо надеяться, теперь вы имеете представление о том, зачем нужен и почему так важен SQL. В следующем разделе будет рассказано об истоках SQL и о поддержке стандартов SQL продуктами Oracle.

## Краткая история SQL

В начале семидесятых годов двадцатого века исследователь из IBM, доктор Е. Ф. Кодд (Codd E. F.) попытался применить строгий математический подход к еще неизведанному тогда миру хранения и извлечения данных. Работа Кодда привела к определению *реляционной модели данных* (*relational data model*) и появлению языка DSL/Alpha для манипулирования данными в реляционной БД. Компании IBM понравилось увиденное, и они открыли проект под названием System/R для построения прототипа на основе работы Кодда. Помимо всего прочего команда System/R разработала упрощенную версию DSL, названную SQUARE, затем переименованную в SEQUEL и, наконец, – в SQL.

Результатом работы, проделанной в рамках System/R, стал выпуск различных продуктов IBM на базе реляционной модели. Под реляционным флагом сплотились и многие другие компании, в том числе и Oracle. К середине 80-х годов язык SQL приобрел на рынке достаточный вес для того, чтобы обратить на себя внимание Американского национального института стандартов (ANSI, American National Standards Institute). В 1986 году ANSI выпустил первый стандарт SQL, затем последовали обновления в 1989, 1992 и 1999 годах.

Спустя тридцать лет после того, как команда System/R начала работать с прототипом реляционной базы данных, SQL все еще занимает устойчивое положение на рынке. Было много попыток свержения реляционных баз данных. Несмотря на это хорошо спроектированные реляционные базы данных вместе с правильно написанными операторами SQL успешно применяются для решения задач обработки больших объемов сложных данных, в то время как другие методы терпят неудачу.

## Реализация SQL от Oracle

Зная, что Oracle рано встала на сторону реляционной модели и SQL, можно подумать, что компанией было приложено немало усилий для достижения соответствия стандартам ANSI. Однако продолжительное время парням из Oracle казалось достаточным, что их реализация SQL была функционально эквивалентна ANSI-стандартам, и полное соответствие их не очень волновало. Начиная же с версии Oracle8i компания стала работать над соответствием ANSI, и в инструментарии Oracle появились оператор CASE и новый синтаксис левого/правого/полного внешнего объединения.

Ирония в том, что, похоже, бизнес-сообщество движется в противоположном направлении. Несколько лет назад все очень заботились о пе-

реносимости и требовали от разработчиков ANSI-совместимого SQL, который бы обеспечивал возможность реализации систем на разных процессорах баз данных. Сегодня же компании выбирают какой-либо процессор базы данных, который будет использоваться в пределах предприятия, и разрешают своим разработчикам применять полный спектр возможностей, не думая об ANSI-совместимости. Одной из причин такого поворота событий служит появление многоуровневых архитектур, где весь доступ к базе данных можно заложить на одном уровне, а не разбрасывать по всему приложению. Другой возможной причиной может быть то, что за последние пять лет на рынке СУБД появились явные лидеры, так что менеджеры ощущают меньший риск при выборе процессора базы данных.

## Теоретическая и практическая терминологии

Штудирова различные труды по реляционной модели данных, вы можете встретить терминологию, которая не будет использоваться в этой книге (например, такие понятия, как *отношения (relations)* или *кортежи (tuples)*). Мы будем применять практические термины, например таблицы и строки. Говоря же о различных частях оператора SQL, будем называть их по имени, а не по выполняемой функции (например, «инструкция SELECT», а не *проекция (projection)*). При всем уважении к доктору Кодду надо сказать, что слово *кортеж* в деловых кругах не произносят, а так как эта книга написана для тех, кто применяет продукты Oracle для решения задач бизнеса, *кортежей* вы не встретите.

## Простая база данных

Данная книга является практическим руководством, поэтому в ней много примеров. Чтобы не придумывать в каждом разделе каждой главы новые наборы таблиц и столбцов, мы решили построить одну простую схему, которая будет применяться в большей части примеров. Выбранной для моделирования предметной областью является работа дистрибьютора (например, оптового продавца автомобильных запчастей или поставщика медицинского оборудования). В этом бизнесе от покупателей поступают заказы на детали, предоставляемые внешними поставщиками. На рис. 1.1 показана модель «объект-отношение» для такого вида деятельности.

Приведем краткое описание модели объект-отношение для тех, кто не знаком с ней. Каждый прямоугольник в модели – это *объект (entity)*, соответствующий таблице<sup>1</sup> в базе данных. Линии между объектами

<sup>1</sup> В зависимости от назначения модели данных объекты могут соответствовать или не соответствовать таблицам базы данных. Например, *логическая модель* описывает бизнес-объекты и их отношения, в то время как *физическая модель* иллюстрирует таблицы и их первичные/внешние ключи. Модель на рис. 1.1 – это физическая модель.



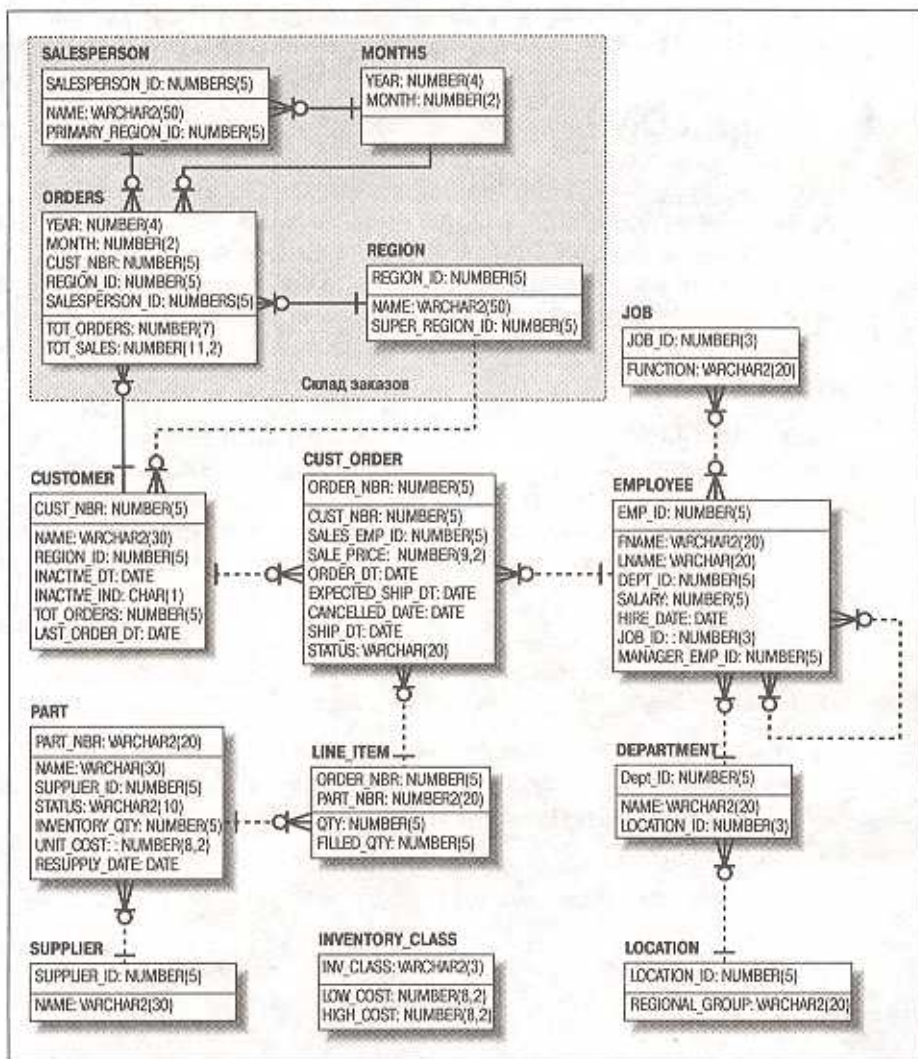


Рис. 1.1. Модель дистрибьютора деталей

представляют отношения (*relationships*) между таблицами, соответствующие внешним ключам. Например, таблица **CUST\_ORDER** (заказы покупателей) содержит внешний ключ для таблицы **EMPLOYEE** (сотрудники), указывающий на продавца, ответственного за определенный заказ. Практически это означает, что таблица **CUST\_ORDER** содержит столбец с идентификаторами (ID) сотрудников и что для каждого заказа этот идентификатор указывает на сотрудника, обработавшего данный заказ. Если вам что-то непонятно, используйте схему просто как иллюстрацию таблиц и столбцов, встречающихся в нашей базе данных.

По мере работы с примерами возвращайтесь время от времени к этой диаграмме, и вы увидите, что начинаете разбираться в отношениях.

## Операторы DML

В этом разделе вы познакомитесь с четырьмя операторами DML. Представленной информации должно быть достаточно для того, чтобы начать писать операторы DML. Однако кажущаяся простота DML обманчива (поговорим об этом в конце раздела), так что помните, что у DML множество аспектов, и лишь о немногих из них мы здесь поговорим.

## Оператор SELECT

Оператор SELECT используется для извлечения данных из базы. Множество данных, извлекаемое оператором SELECT, называется *результатирующим множеством (result set)*. Как и таблица, результирующее множество состоит из строк и столбцов, что позволяет заполнить таблицу данными результирующего множества. Общий вид оператора SELECT таков:

```
SELECT <один или несколько объектов>
FROM <одно или несколько мест>
WHERE <ни одного, одно или несколько условий>
```

Инструкции SELECT и FROM необходимы, а вот инструкция WHERE необязательна (хотя и она почти всегда используется). Начнем с простого примера, извлекающего три столбца из каждой строки таблицы CUSTOMER (заказчики):

```
SELECT cust_nbr, name, region_id
FROM customer;
```

CUST_NBR	NAME	REGION_ID
1	Cooper Industries	5
2	Emblazon Corp.	5
3	Ditech Corp.	5
4	Flowtech Inc.	5
5	Gentech Industries	5
6	Spartan Industries	6
7	Wallace Labs	6
8	Zantech Inc.	6
9	Cardinal Technologies	6
10	Flowrite Corp.	6
11	Glaven Technologies	7
12	Johnson Labs	7
13	Kimball Corp.	7
14	Madden Industries	7
15	Turntech Inc.	7

16 Paulson Labs	8
17 Evans Supply Corp.	8
18 Spalding Medical Inc.	8
19 Kendall-Taylor Corp.	8
20 Malden Labs	8
21 Crimson Medical Inc.	9
22 Nichols Industries	9
23 Owens-Baxter Corp.	9
24 Jackson Medical Inc.	9
25 Worcester Technologies	9
26 Alpha Technologies	10
27 Phillips Labs	10
28 Jaztech Corp.	10
29 Madden-Taylor Inc.	10
30 Wallace Industries	10

Инструкция WHERE не была использована, и никаких ограничений на данные мы не наложили, поэтому запрос возвращает все строки таблицы заказчиков. Если необходимо ограничить возвращаемый набор данных, можно добавить в оператор инструкции WHERE с одним условием:

```
SELECT cust_nbr, name, region_id
FROM customer
WHERE region_id = 8;
```

CUST_NBR	NAME	REGION_ID
-----		
16	Paulson Labs	8
17	Evans Supply Corp.	8
18	Spalding Medical Inc.	8
19	Kendall-Taylor Corp.	8
20	Malden Labs	8

Теперь результирующее множество содержит только заказчиков, проживающих в области с идентификатором region\_id, равным 8. Но что если нужно сослаться на область по имени, а не по номеру? Можно выбрать нужное имя из таблицы REGION, а затем, зная region\_id, обратиться к таблице CUSTOMER. Чтобы не писать два разных запроса, можно получить тот же результат с помощью одного запроса, использующего *объединение (join)*:

```
SELECT customer.cust_nbr, customer.name, region.name
FROM customer, region
WHERE region.name = 'New England'
AND region.region_id = customer.region_id;
```

CUST_NBR	NAME	NAME
-----		
1	Cooper Industries	New England
2	Emblazon Corp.	New England
3	Oltech Corp.	New England
4	Flowtech Inc.	New England
5	Gentech Industries	New England



Теперь в инструкции FROM не одна таблица, а две, и инструкция WHERE содержит *условие объединения (join condition)*, которое указывает, что таблицы заказчиков и областей должны быть объединены по столбцу *region\_id*, имеющемуся в каждой таблице. Объединения и их условия будут детально рассмотрены в главе 3.

Так как обе таблицы содержат столбец с названием *name*, необходимо как-то определить, какой именно столбец вас интересует. В предыдущем примере это делается с помощью точечной нотации – добавления через точку имени таблицы перед именем столбца. Если же на написание полных имен таблиц у вас уходит слишком много времени, назначьте для каждого названия таблицы в инструкции FROM *псевдоним (alias)* и используйте его вместо имени в инструкциях SELECT и WHERE:

```
SELECT c.cust_nbr, c.name, r.name
FROM customer c, region r
WHERE r.name = 'New England'
AND r.region_id = c.region_id;
```

В этом примере псевдоним «с» был присвоен таблице заказчиков, а псевдоним «r» – таблице областей. Теперь можно в инструкциях SELECT и WHERE писать «с» вместо «customer» и «r» вместо «region».

## Элементы инструкции SELECT

Рассмотренные ранее результирующие множества, порожденные нашими запросами, содержали столбцы одной или нескольких таблиц. Как правило, элементами инструкции SELECT действительно являются ссылки на столбцы; среди них также могут встречаться:

- Константы, такие как числа (1) или строки ('abc')
- Выражения, например `shape.diameter * 3.1415927`
- Функции, такие как `TO_DATE('01-JAN-2002', 'DD-MON-YYYY')`
- Псевдостолбцы, например ROWID, ROWNUM или LEVEL

Если первые три пункта в списке довольно просты, то последний нуждается в пояснении. В Oracle есть несколько столбцов-призраков, называемых *псевдостолбцами (pseudocolumns)*, которые не присутствуют ни в одной таблице. Их значения появляются во время выполнения запроса, и в некоторых ситуациях они бывают полезны.

Например, псевдостолбец ROWID представляет физическое положение строки, обеспечивая быстрейший механизм доступа к строкам. Он может оказаться полезным, если вы планируете удалить или обновлять полученные в результате запроса строки. Но никогда не храните значения ROWID в базе данных и не ссылайтесь на них за пределами транзакции, в которой они были получены, так как в определенных ситуациях значение ROWID для строки может поменяться, а при удалении строки ее значение ROWID может перейти к другой.



Приведем пример, демонстрирующий использование всех элементов из списка:

```
SELECT rownum,  
       cust_nbr,  
       1 multiplier,  
       'cust # ' || cust_nbr cust_nbr_str,  
       'hello' greeting,  
       TO_CHAR(last_order_dt, 'DD-MON-YYYY') last_order  
FROM customer;
```

ROWNUM	CUST_NBR	MULTIPLIER	CUST_NBR_STR	GREETING	LAST_ORDER
1	1	1	cust # 1	hello	15-JUN-2000
2	2	1	cust # 2	hello	27-JUN-2000
3	3	1	cust # 3	hello	07-JUL-2000
4	4	1	cust # 4	hello	15-JUL-2000
5	5	1	cust # 5	hello	01-JUN-2000
6	6	1	cust # 6	hello	10-JUN-2000
7	7	1	cust # 7	hello	17-JUN-2000
8	8	1	cust # 8	hello	22-JUN-2000
9	9	1	cust # 9	hello	25-JUN-2000
10	10	1	cust # 10	hello	01-JUN-2000
11	11	1	cust # 11	hello	05-JUN-2000
12	12	1	cust # 12	hello	07-JUN-2000
13	13	1	cust # 13	hello	07-JUN-2000
14	14	1	cust # 14	hello	05-JUN-2000
15	15	1	cust # 15	hello	01-JUN-2000
16	16	1	cust # 16	hello	31-MAY-2000
17	17	1	cust # 17	hello	28-MAY-2000
18	18	1	cust # 18	hello	23-MAY-2000
19	19	1	cust # 19	hello	16-MAY-2000
20	20	1	cust # 20	hello	01-JUN-2000
21	21	1	cust # 21	hello	26-MAY-2000
22	22	1	cust # 22	hello	18-MAY-2000
23	23	1	cust # 23	hello	08-MAY-2000
24	24	1	cust # 24	hello	26-APR-2000
25	25	1	cust # 25	hello	01-JUN-2000
26	26	1	cust # 26	hello	21-MAY-2000
27	27	1	cust # 27	hello	08-MAY-2000
28	28	1	cust # 28	hello	23-APR-2000
29	29	1	cust # 29	hello	06-APR-2000
30	30	1	cust # 30	hello	01-JUN-2000

Интересно, что в инструкции SELECT совсем необязательно ссылаться на столбцы таблиц инструкции FROM. Например, результирующее множество следующего запроса целиком построено из констант:

```
SELECT 1 num, 'abc' str  
FROM customer;
```





3 Ditech Corp.	New England
2 Emblazon Corp.	New England
4 Flowtech Inc.	New England
5 Gentech Industries	New England

Сортируемые столбцы можно определять по их положению в инструкции SELECT. Отсортируем предыдущий запрос по столбцу CUST\_NBR (номер заказчика), который в инструкции SELECT указан первым:

```
SELECT c.cust_nbr, c.name, r.name
FROM customer c, region r
WHERE r.name = 'New England'
      AND r.region_id = c.region_id
ORDER BY 1;
```

CUST_NBR	NAME	NAME
1	Cooper Industries	New England
2	Emblazon Corp.	New England
3	Ditech Corp.	New England
4	Flowtech Inc.	New England
5	Gentech Industries	New England

Указание ключей сортировки по позиции сэкономит вам немного времени, но может привести к ошибкам, если в дальнейшем порядок следования столбцов в инструкции SELECT будет изменен.

## Удаление дубликатов

В некоторых случаях результирующее множество может содержать одинаковые данные. Например, при формировании списка проданных за последний месяц деталей те детали, которые присутствовали в нескольких заказах, встретятся в результирующем множестве несколько раз. Чтобы исключить повторы, вставьте в инструкцию SELECT ключевое слово DISTINCT:

```
SELECT DISTINCT li.part_nbr
FROM cust_order co, line_item li
WHERE co.order_dt >= TO_DATE('01-JUL-2001', 'DD-MON-YYYY')
      AND co.order_dt < TO_DATE('01-AUG-2001', 'DD-MON-YYYY')
      AND co.order_nbr = li.order_nbr;
```

Запрос возвращает множество отличающихся друг от друга записей о деталях, заказанных в течение июля 2001 года. Без ключевого слова DISTINCT вывод содержал бы по одной строке для каждой строки каждого заказа, и одна деталь могла бы встречаться несколько раз, если бы она содержалась в нескольких заказах. Принимая решение о включении DISTINCT в инструкцию SELECT, следует иметь в виду, что поиск и удаление дубликатов требует сортировки, которая будет служить дополнительной нагрузкой при выполнении запроса.

## Оператор INSERT

Оператор INSERT — это механизм загрузки данных в базу данных. За один раз данные можно вставить только в одну таблицу, зато брать вставляемые данные можно из нескольких дополнительных таблиц. Вставляя данные в таблицу, не нужно указывать значения для каждого столбца, однако следует обратить внимание на то, допускают ли столбцы использование значений NULL<sup>1</sup> или же нет. Посмотрим на определение таблицы EMPLOYEE (служащие):

```
describe employee
```

Name	Null?	Type
EMP_ID	NOT NULL	NUMBER(5)
FNAME		VARCHAR2(20)
LNAME	NOT NULL	VARCHAR2(20)
DEPT_ID	NOT NULL	NUMBER(5)
MANAGER_EMP_ID		NUMBER(5)
SALARY		NUMBER(5)
HIRE_DATE		DATE
JOB_ID		NUMBER(3)

Пометка NOT NULL для столбцов emp\_id, lname и dept\_id означает, что они обязательны для заполнения. Поэтому в операторе INSERT необходимо указать значения как минимум для трех этих столбцов, например, как показано ниже:

```
INSERT INTO employee (emp_id, lname, dept_id)
VALUES (101, 'Smith', 2);
```

Количество элементов в инструкции VALUES должно совпадать с количеством элементов в списке столбцов, а их типы данных должны соответствовать определениям столбцов. В нашем примере emp\_id и dept\_id хранят численные значения, а lname — строковое, поэтому оператор INSERT выполнится без ошибок. Oracle всегда автоматически пытается преобразовать один тип данных в другой, поэтому следующий оператор тоже выполнится без ошибок:

```
INSERT INTO employee (emp_id, lname, dept_id)
VALUES ('101', 'Smith', '2');
```

Иногда вставляемые данные нужно предварительно извлечь из одной или нескольких таблиц. Так как оператор SELECT формирует результирующее множество, состоящее из строк и столбцов, то можно непосредственно передать его оператору INSERT:

```
INSERT INTO employee (emp_id, fname, lname, dept_id, hire_date)
SELECT 101, 'Dave', 'Smith', d.dept_id, SYSDATE
```

---

<sup>1</sup> NULL означает отсутствие значения; подробно обсуждается в главе 2.



```
FROM department d
WHERE d.name = 'Accounting';
```

В данном примере оператор SELECT извлекает идентификатор отдела для бухгалтерии (Accounting). Остальные четыре столбца в операторе SELECT представлены константами.

## Оператор DELETE

Оператор DELETE обеспечивает удаление данных из базы. Как и SELECT, оператор DELETE содержит инструкцию WHERE с условиями для идентификации удаляемых строк. Забыв указать инструкцию WHERE в операторе DELETE, вы удалите все строки из указанной таблицы. Следующий оператор удаляет всех сотрудников с фамилией Hooper из таблицы EMPLOYEE:

```
DELETE FROM employee
WHERE lname = 'Hooper';
```

Иногда значения, необходимые для построения условия в инструкции WHERE, располагаются в других таблицах. Например, решение компании вынести вонне функции бухучета потребует удаления всего бухгалтерского персонала из таблицы EMPLOYEE:

```
DELETE FROM employee
WHERE dept_id =
  (SELECT dept_id
   FROM department
   WHERE name = 'Accounting');
```

Подобное использование оператора SELECT носит название *подзапроса (subquery)* и будет изучено в главе 5.

## Оператор UPDATE

С помощью оператора UPDATE вносятся изменения в существующие данные. Как и DELETE, оператор UPDATE включает в себя инструкцию WHERE для указания тех строк, которые будут изменены. Посмотрим, как можно предоставить 10-процентное повышение зарплаты тем, у кого годовой доход меньше 40 000 долларов:

```
UPDATE employee
SET salary = salary * 1.1
WHERE salary < 40000;
```

Если необходимо изменить несколько столбцов, вы можете выбрать один из двух вариантов: задать набор пар столбец-значение, разделенных запятыми, или указать набор столбцов и подзапрос. Два следующих оператора UPDATE изменяют столбцы inactive\_dt и inactive\_ind в

таблице `CUSTOMER` для клиентов, не сделавших ни одного заказа за последний год:

```
UPDATE customer
SET inactive_dt = SYSDATE, inactive_ind = 'Y'
WHERE last_order_dt < SYSDATE - 365;

UPDATE customer
SET (inactive_dt, inactive_ind) =
  (SELECT SYSDATE, 'Y' FROM dual)
WHERE last_order_dt < SYSDATE - 365;
```

Подзапрос во втором примере выглядит немного неестественно, так как он обращается к таблице `dual`<sup>1</sup> для построения результирующего множества, состоящего из двух констант; он приведен для иллюстрации использования подзапросов в операторе `UPDATE`. В следующих главах вы найдете намного более интересные примеры использования подзапросов.

## И зачем остальные 13 глав?

Прочтя эту главу, вы могли подумать, что `SQL` (или, по крайней мере, `DML`) чрезвычайно прост. Да, на поверхности он кажется простым, и вы уже знаете об этом языке достаточно, чтобы писать программы. Но со временем вы поймете, что существует множество путей, ведущих к одной и той же цели, и некоторые из них гораздо эффективнее и элегантнее, чем другие. Настоящий мастер `SQL` может разом стереть весь написанный за год код и переделать его заново. Один из нас шел к этому девять лет. Надеемся, что эта книга поможет сократить ваш путь.

Читая оставшуюся часть книги, вы заметите, что большинство примеров — это операторы `SELECT`, а остаток практически равномерно распределен между операторами `INSERT`, `UPDATE` и `DELETE`. Такое неравенство не означает, что `SELECT` важнее других операторов `DML`. Операторы `SELECT` преобладают, потому что есть возможность показать результат выполнения, что помогает лучше понять запрос. К тому же, многие приемы, использованные в операторах `SELECT`, также применимы и к операторам `UPDATE` и `DELETE`.

---

<sup>1</sup> `Dual` — это специальная таблица `Oracle`, состоящая ровно из одной строки и одного столбца. Она бывает полезной, когда необходимо построить запрос, возвращающий ровно одну строку.



# 2

## Инструкция WHERE

Инструкция WHERE – это тот механизм, который необходим для идентификации набора данных при организации запроса, изменении или удалении данных. В этой главе будет рассмотрена роль инструкции WHERE в операторах SQL, а также различные варианты построения этой инструкции.

### Жизнь без WHERE

Прежде чем приступить к исследованию инструкции WHERE, давайте попробуем представить, что ее не существует. Пусть, например, речь идет о работе с таблицей деталей (part). Чтобы получить данные из этой таблицы, создадим следующий запрос:

```
SELECT part_nbr, name, supplier_id, status, inventory_qty  
FROM part;
```

Если таблица деталей содержит 10 000 элементов, то возвращенное запросом результирующее множество будет состоять из 10 000 строк, в каждой из которых 5 столбцов. Затем предстоит загрузить эти 10 000 строк в оперативную память и произвести необходимые изменения.

После того как данные, находящиеся в оперативной памяти, изменены, необходимо внести эти изменения в таблицу деталей. Указать, какие строки необходимо изменить, невозможно, так что остается только удалить все строки таблицы и заново вставить все 10 000 строк:

```
DELETE FROM part;  
  
INSERT INTO part (part_nbr, name, supplier_id, status, inventory_qty)  
VALUES ('XY5-1002', 'Wonder Widget', 1, 'IN-STOCK', 1);  
  
/* Остальные 9 999 операторов INSERT */
```

Теоретически такой подход возможен, но он губителен для производительности, параллелизма (возможности одновременного изменения данных несколькими пользователями) и масштабируемости.

Пусть теперь необходимо изменить только те данные таблицы, которые относятся к деталям одного поставщика – Acme Industries. Имена поставщиков хранятся в таблице `supplier`, поэтому в инструкции `FROM` следует указать обе таблицы: деталей (`part`) и поставщиков (`supplier`):

```
SELECT p.part_nbr, p.name, p.supplier_id, p.status, p.inventory_qty,  
       s.supplier_id, s.name  
FROM part p, supplier s;
```

Если 100 компаний поставляют 10 000 деталей (таблица `part`), то такой запрос вернет 1 000 000 строк. Это число, называемое прямым, или декартовым, произведением (*Cartesian product*), равно количеству всех возможных комбинаций строк двух таблиц. «Просеивая» миллион строк, будем искать те, в которых значения `p.supplier_id` и `s.supplier_id` равны и значение столбца `s.name` равно 'Acme Industries'. Если фирма Acme Industries поставляет всего 50 из 10 000 деталей, представленных в базе данных, нам предстоит отбросить 999 950 из 1 000 000 строк, возвращенных запросом.

## На помощь приходит WHERE

Эти примеры наглядно иллюстрируют полезность инструкции `WHERE`, которая предоставляет возможность:

1. Отфильтровывать ненужные данные из результирующего множества.
2. Изолировать одну или несколько строк таблицы для изменения.
3. Выполнить условное объединение двух или более наборов данных.

Чтобы посмотреть, как все это работает, давайте добавим инструкцию `WHERE` в написанный ранее оператор `SELECT`, который занимается поиском всех деталей, поставляемых Acme Industries:

```
SELECT p.part_nbr, p.name, p.supplier_id, p.status, p.inventory_qty,  
       s.supplier_id, s.name  
FROM part p, supplier s  
WHERE s.supplier_id = p.supplier_id  
      AND s.name = 'Acme Industries';
```

В данном случае инструкция `WHERE` состоит из двух частей, называемых *условиями* (*conditions*), которые вычисляются и оцениваются по отдельности. Значения условий всегда равны или `TRUE`, или `FALSE`. Если в инструкции `WHERE` входит несколько условий, то для того чтобы некоторая строка была включена в результирующее множество, значения всех условий для нее должны быть равны `TRUE`.<sup>1</sup> В рассматриваемом



примере строка, образованная при объединении данных из таблиц `part` и `supplier`, будет включена в окончательное результирующее множество только в том случае, если обе таблицы имеют одинаковое значение столбца `supplier_id` и значение столбца `name` таблицы `supplier` совпадает с `'Acme Industries'`.<sup>1</sup> Любые другие сочетания данных из двух таблиц будут оценены как `FALSE` и отброшены.

С добавлением в рассмотренный ранее пример инструкции `WHERE` работу по отсеиванию ненужных строк результирующего множества берет на себя Oracle, и запрос вернет всего 50, а не 1 000 000 строк. После того как нужные 50 строк извлечены из базы данных, можно приступить к изменению данных. Помните, что теперь, когда в вашем распоряжении есть инструкция `WHERE`, больше не нужно удалять и вновь вставлять измененные данные, вместо этого можно просто использовать оператор `UPDATE` для изменения отдельных строк на основе значения столбца `part_nbr`, который является уникальным идентификатором строки таблицы:

```
UPDATE part
SET status = 'DISCONTINUED'
WHERE part_nbr = 'A15-4557';
```

Хотя это явное усовершенствование, можно пойти и дальше. Если планируется изменить статус для всех 50 деталей, поставленных Acme Industries, то в запросе вообще нет необходимости. Просто выполняем оператор `UPDATE`, который находит и изменяет все 50 записей:

```
UPDATE part
SET status = 'DISCONTINUED'
WHERE supplier_id =
  (SELECT supplier_id
   FROM supplier
   WHERE name = 'Acme Industries');
```

Инструкция `WHERE` этого оператора состоит из одного условия, которое определяет равенство столбца `supplier_id` значению, возвращаемому запросом к таблице `supplier`. Запрос, заключенный в скобки внутри другого SQL-оператора, называется *подзапросом (subquery)*; о подзапросах будет подробно рассказано в главе 5, пока же просто не пугай-

---

<sup>1</sup> Это чрезмерно упрощенное пояснение. Далее вы узнаете, что использование операторов `OR` и `NOT` может привести к тому, что значение инструкции `WHERE` будет вычислено как `TRUE` несмотря на то, что отдельные условия оцениваются как `FALSE`.

<sup>1</sup> Еще одно упрощение. Оптимизатор Oracle (компонент, задача которого состоит в поиске наиболее эффективного способа выполнения запроса) не создает все возможные комбинации строк всех таблиц или представлений, входящих в инструкцию `FROM` прежде, чем вычислять условия. Оптимизатор выбирает порядок, в котором следует вычислять условия и объединять данные, чтобы время исполнения было минимальным.



тесь их. В результате условие будет переписано с использованием величины, возвращенной подзапросом, например:

```
UPDATE part
SET status = 'DISCONTINUED'
WHERE supplier_id = 1;
```

Условие оценивается как TRUE для 50 из 10 000 строк таблицы part, и значение поля status для этих 50 строк меняется на 'DISCONTINUED'.

## Вычисление инструкции WHERE

Вы видели, как работает инструкция WHERE, теперь давайте поговорим о том, как она вычисляется. Уже упоминалось, что инструкция WHERE состоит из одного или нескольких условий, которые независимо друг от друга оцениваются как TRUE или FALSE. Если инструкция WHERE состоит из нескольких условий, такие условия разделяются логическими операторами AND и OR. В зависимости от значения каждого отдельного условия и размещения между ними логических операторов Oracle присваивает соответствующее конечное значение TRUE или FALSE каждой «испытываемой» строке, определяя тем самым, войдет ли данная строка в итоговое результирующее множество.

Давайте вернемся к запросу об Acme Industries:

```
SELECT p.part_nbr, p.name, p.supplier_id, p.status, p.inventory_qty,
       s.supplier_id, s.name
FROM part p, supplier s
WHERE s.supplier_id = p.supplier_id
AND s.name = 'Acme Industries';
```

Инструкция WHERE состоит из двух условий, разделенных оператором AND. Поэтому строка будет включена в результирующее множество, только если оба условия будут вычислены как TRUE. Варианты оценки нескольких условий, разделенных AND, приведены в табл. 2.1 (условия заменены своими возможными значениями TRUE и FALSE).

*Таблица 2.1. Вычисление нескольких условий, разделенных оператором AND*

Промежуточный результат	Конечный результат
WHERE TRUE AND TRUE	TRUE
WHERE FALSE AND FALSE	FALSE
WHERE FALSE AND TRUE	FALSE
WHERE TRUE AND FALSE	FALSE

Используя основные логические правила, устанавливаем, что единственная комбинация результатов оценки отдельных условий, итоговое

значение которой для исследуемой строки равно TRUE, возникает в том случае, когда оба условия вычислены как TRUE. Варианты результатов для условий, разделенных оператором OR, приведены в табл. 2.2.

*Таблица 2.2. Вычисление нескольких условий, разделенных оператором OR*

Промежуточный результат	Конечный результат
WHERE TRUE OR TRUE	TRUE
WHERE FALSE OR FALSE	FALSE
WHERE FALSE OR TRUE	TRUE
WHERE TRUE OR FALSE	TRUE

Давайте теперь немного оживим наш запрос, включив в него детали, как поставляемые Acme Industries, так и полученные от Tilton Enterprises:

```
SELECT p.part_nbr, p.name, p.supplier_id, p.status, p.inventory_qty,
       s.supplier_id, s.name
FROM part p, supplier s
WHERE s.supplier_id = p.supplier_id
      AND (s.name = 'Acme Industries'
           OR s.name = 'Tilton Enterprises');
```

Теперь в инструкции WHERE три условия, разделенные AND и OR, при этом два условия заключены в скобки. Возможные результаты оценки приведены в табл. 2.3.

*Таблица 2.3. Вычисление нескольких условий, разделенных операторами OR и AND*

Промежуточный результат	Конечный результат
WHERE TRUE AND (TRUE OR FALSE)	TRUE
WHERE TRUE AND (FALSE OR TRUE)	TRUE
WHERE TRUE AND (FALSE OR FALSE)	FALSE
WHERE FALSE AND (TRUE OR FALSE)	FALSE
WHERE FALSE AND (FALSE OR TRUE)	FALSE
WHERE FALSE AND (FALSE OR FALSE)	FALSE

Так как одна деталь не может быть получена одновременно и от Acme Industries и от Tilton Enterprises, такие промежуточные результаты, как TRUE AND (TRUE AND TRUE) и FALSE AND (TRUE AND TRUE), не рассматриваются.

Чтобы сделать все еще интереснее, введем оператор NOT. Следующий запрос возвращает данные о деталях, полученных не от Acme Industries или Tilton Enterprises:



```

SELECT p.part_nbr, p.name, p.supplier_id, p.status, p.inventory_qty,
       s.supplier_id, s.name
FROM part p, supplier s
WHERE s.supplier_id = p.supplier_id
      AND NOT (s.name = 'Acme Industries'
              OR s.name = 'Tilton Enterprises');

```

В табл. 2.4 можно увидеть, как добавление оператора NOT повлияло на вычисляемый результат.

*Таблица 2.4. Вычисление нескольких условий, разделенных операторами OR, AND и NOT*

Промежуточный результат	Конечный результат
WHERE TRUE AND NOT (TRUE OR FALSE)	FALSE
WHERE TRUE AND NOT (FALSE OR TRUE)	FALSE
WHERE TRUE AND NOT (FALSE OR FALSE)	TRUE
WHERE FALSE AND NOT (TRUE OR FALSE)	FALSE
WHERE FALSE AND NOT (FALSE OR TRUE)	FALSE
WHERE FALSE AND NOT (FALSE OR FALSE)	FALSE

Надо сказать, что использование оператора NOT в предыдущем примере несколько надуманно; позже вы узнаете, что существуют более естественные пути выражения той же логики.

## Условия и выражения

Как вычисляются и группируются условия, вы уже знаете, пришло время поговорить об элементах, составляющих условия. Условие состоит из одного или нескольких *выражений* (*expressions*) и одного или нескольких *операторов* (*operators*). К выражениям относятся:

- Числа
- Столбцы, например `supplier_id`
- Константы, например `'Acme Industries'`
- Функции, например `UPPER('abcd')`
- Списки простых выражений, например `(1, 2, 3)`
- Подзапросы

К операторам относятся:

- Арифметические операторы, такие как `+`, `-`, `*` и `/`
- Операторы сравнения, такие как `=`, `<`, `>`, `!=`, `LIKE` и `IN`

В последующих разделах будут рассмотрены наиболее распространенные типы условий, использующие различные комбинации перечисленных выше типов выражений и операторов.



## Условия равенства и неравенства

Чаще других при построении инструкции WHERE используются условия равенства — для объединения наборов данных или выделения определенных значений. Вы уже не раз встречались с такими условиями в рассмотренных ранее запросах, например:

```
s.supplier_id = p.supplier_id
s.name = 'Acme Industries'
supplier_id = (SELECT supplier_id
FROM supplier
WHERE name = 'Acme Industries')
```

Во всех трех примерах за столбцом-выражением следует оператор сравнения (=), после которого стоит второе выражение. Условия отличаются типом выражения, стоящего справа от оператора сравнения. В первом примере один столбец сравнивается с другим, во втором столбец сравнивается с константой, в третьем столбец сравнивается со значением, возвращаемым подзапросом.

Можно построить условие и при помощи оператора неравенства (!=). Ранее для поиска информации о деталях, полученных ото всех поставщиков кроме Acme Industries и Tilton Enterprises, был использован оператор NOT. Если вместо NOT применить оператор !=, запрос станет понятнее и исчезнет необходимость в операторе OR:

```
SELECT p.part_nbr, p.name, p.supplier_id, p.status, p.inventory_qty,
       s.supplier_id, s.name
FROM part p, supplier s
WHERE s.supplier_id = p.supplier_id
      AND s.name != 'Acme Industries'
      AND s.name != 'Tilton Enterprises';
```

Улучшение предыдущей версии налицо, но в следующем разделе будет представлен еще более четкий и ясный способ реализации той же самой логики.

## Условия принадлежности

Не менее полезным, чем проверка идентичности двух выражений, может быть установление принадлежности выражения некоторому множеству выражений. Используя оператор IN, можно построить условие, которое будет оценено как TRUE, если указанное выражение принадлежит заданному множеству выражений:

```
s.name IN ('Acme Industries', 'Tilton Enterprises')
```

Можно также добавить оператор NOT, чтобы определить, что выражение не входит в указанное множество:

```
s.name NOT IN ('Acme Industries', 'Tilton Enterprises')
```

Большинство программистов предпочитает одно условие с использованием IN или NOT IN множеству условий с операторами = или !=, так что давайте нанесем последний удар по запросу Acme/Tilton:

```
SELECT p.part_nbr, p.name, p.supplier_id, p.status, p.inventory_qty,  
       s.supplier_id, s.name  
FROM part p, supplier s  
WHERE s.supplier_id = p.supplier_id  
      AND s.name NOT IN ('Acme Industries', 'Tilton Enterprises');
```

Множества выражений могут быть не только предопределенными, но и генерироваться подзапросами во время выполнения. Если подзапрос возвращает ровно одну строку, можно использовать оператор сравнения, если же возвращается несколько строк или же точно неизвестно, сколько строк будет возвращено, используйте оператор IN. Приведенный ниже пример обновляет все заказы, в которых указаны детали, полученные от Eastern Importers:

```
UPDATE cust_order  
SET sale_price = sale_price *1.1  
WHERE cancelled_dt IS NULL  
      AND ship_dt IS NULL  
      AND order_nbr IN  
      (SELECT li.order_nbr  
       FROM line_item li, part p, supplier s  
       WHERE s.name = 'Eastern Importers'  
             AND s.supplier_id = p.supplier_id  
             AND p.part_nbr = li.part_nbr);
```

Результат выполнения подзапроса – множество номеров заказов (возможно, пустое). Все заказы, чьи номера принадлежат данному множеству, будут изменены оператором UPDATE.

## Условия вхождения значения в диапазон

При работе с датами или числовыми данными может встать вопрос не о равенстве значения какой-то величине и не о его принадлежности конечному множеству, а о вхождении значения в некоторый диапазон. В подобных случаях удобно использовать оператор BETWEEN..AND, например:

```
DELETE FROM cust_order  
WHERE order_dt BETWEEN '01-JUL-2001' AND '31-JUL-2001';
```

Чтобы узнать, расположено ли значение вне какого-то диапазона, добавим оператор NOT:

```
SELECT order_nbr, cust_nbr, sale_price  
FROM cust_order  
WHERE sale_price NOT BETWEEN 1000 AND 10000;
```

Первое значение, указываемое в BETWEEN, должно быть меньше второго. Несмотря на то что выражения «BETWEEN 1 AND 10» и «BETWEEN 10 AND 1»



могут показаться логически эквивалентными, значение второго всегда будет равно FALSE, так как первым указано большее значение.

Диапазоны можно задавать при помощи операторов <, >, <= и >=, но в этом случае придется указывать два условия вместо одного. Предыдущий запрос можно записать следующим образом:

```
SELECT order_nbr, cust_nbr, sale_price
FROM cust_order
WHERE sale_price < 1000 OR sale_price > 10000;
```

## Условия совпадения

Если вы имеете дело с символьными данными, бывают ситуации, когда важно полное совпадение строк, иногда же достаточно частичного совпадения. В последнем случае можно применить оператор LIKE, указав один или несколько символов для поиска совпадения с образцом, например:

```
DELETE FROM part
WHERE part_nbr LIKE 'ABC%';
```

Специальный символ «%» совпадает со строками любой длины, поэтому удалены будут записи о деталях 'ABC', 'ABC-123', 'ABC99999999' и т. д. Если необходим более точный контроль, можно использовать специальный символ подчеркивания (\_), который замещает любой одиночный символ, например:

```
DELETE FROM part
WHERE part_nbr LIKE '_B_';
```

Такому образцу соответствуют номера деталей, состоящие ровно из трех символов, в середине которых стоит B – они и будут удалены. Специальные символы могут использоваться в различных комбинациях. Кроме того, для поиска строк, не совпадающих с заданным образцом, применяется оператор NOT. В приведенном ниже примере удаляются все детали, названия которых не содержат буквы Z в третьей позиции, за которой в произвольном месте следовала бы строка «T1J»:

```
DELETE FROM part
WHERE part_nbr NOT LIKE '_Z%T1J%';
```

Oracle предоставляет для обработки символьных данных множество встроенных функций, которые могут быть использованы при построении условий совпадения с образцом. Например, условие `part_nbr LIKE 'ABC%'` можно переписать, используя функцию SUBSTR следующим образом: `SUBSTR(part_nbr, 1, 3) = 'ABC'`. Описания и примеры использования встроенных функций Oracle приведены в книге «Oracle SQL: Essential Reference» (O'Reilly).



## Обработка выражения NULL

Выражение NULL представляет собой отсутствие значения. Если при вводе заказа в базу данных вы не имеете достоверной информации о дате его отгрузки, лучше оставить эту дату неопределенной, чем придумывать значение. До тех пор пока дата отправки не станет известна, лучше поставить в столбец ship\_dt значение NULL. NULL также удобно использовать в тех случаях, когда другое значение неприменимо. Например, дата отправки отмененного заказа более не имеет смысла, поэтому следует установить ее равной NULL.

При работе с NULL термин «равенство» непригоден, столбец может *быть* NULL, но он никогда не *равен* NULL. Поэтому для выбора NULL-значений необходимо использовать специальный оператор IS, например:

```
UPDATE cust_order
SET expected_ship_dt = SYSDATE + 1
WHERE ship_dt IS NULL;
```

В данном примере для всех заказов, для которых не была определена дата отправки, дата предполагаемой отправки будет сдвинута на день вперед от текущей даты.

Для поиска не NULL-значений можно использовать оператор NOT:

```
UPDATE cust_order
SET expected_ship_dt = NULL
WHERE ship_dt IS NOT NULL;
```

В этом примере для всех уже отправленных заказов дата ожидаемой отправки устанавливается в NULL. Обратите внимание на то, что инструкция SET применяет оператор равенства (=) для NULL, в то время как инструкция WHERE использует операторы IS и NOT. Оператор равенства применяется для присваивания столбцу значения NULL, а оператор IS определяет, содержится ли в столбце NULL. Если бы создатели Oracle выбрали специальный оператор для установления значения столбца в NULL (например, SET expected\_ship\_dt TO NULL), удалось бы избежать огромного количества ошибок – но этого не случилось. Чтобы еще более все усложнить, Oracle не выражает недовольства при ошибочном использовании оператора равенства для NULL. Приведенный ниже запрос будет проанализирован и выполнен, но не будет возвращать строки:

```
SELECT order_nbr, cust_nbr, sale_price, order_dt
FROM cust_order
WHERE ship_dt = NULL;
```

Будем надеяться, вы быстро заметите, что запрос никогда не возвращает данные, и замените оператор равенства на оператор IS. Но есть и гораздо менее очевидная ошибка, относящаяся к работе с NULL, которую сложнее распознать. Пусть необходимо найти всех сотрудников,

которыми не руководит Jeff Blake, чей идентификатор служащего равен 11. Инстинктивно хочется написать такой запрос:

```
SELECT fname, lname, manager_emp_id
FROM employee
WHERE manager_emp_id != 11;
```

FNAME	LNAME	MANAGER_EMP_ID
Alex	Fox	28
Chris	Anderson	28
Lynn	Nichols	28
Eric	Iverson	28
Laura	Peters	28
Mark	Russell	28

Хотя этот запрос и возвращает строки, он упускает тех служащих, которые являются топ-менеджерами, то есть у них нет руководителя. Так как значение NULL несопоставимо с числом 11, это множество сотрудников отсутствует в результирующем множестве. Чтобы убедиться в том, что в рассмотрение принимаются все служащие, необходимо обрабатывать NULL явным образом:

```
SELECT fname, lname, manager_emp_id
FROM employee
WHERE manager_emp_id IS NULL OR manager_emp_id != 11;
```

FNAME	LNAME	MANAGER_EMP_ID
Bob	Brown	
John	Smith	
Jeff	Blake	
Alex	Fox	28
Chris	Anderson	28
Lynn	Nichols	28
Eric	Iverson	28
Laura	Peters	28
Mark	Russell	28

Включение двух условий в инструкцию WHERE может показаться не слишком интересным. Вместо этого можно использовать встроенную функцию Oracle NVL, которая подставляет указанное значение для столбцов, содержащих NULL:

```
SELECT fname, lname, manager_emp_id
FROM employee
WHERE NVL(manager_emp_id, -999) != 11;
```

FNAME	LNAME	MANAGER_EMP_ID
Bob	Brown	
John	Smith	



Jeff	Blake	
Alex	Fox	28
Chris	Anderson	28
Lynn	Nichols	28
Eric	Iverson	28
Laura	Peters	28
Mark	Russell	28

В данном примере значение -999 заменяет все значения NULL, а так как -999 никогда не равно 11, тем самым гарантируется, что все строки, в которых столбец `manager_emp_id` содержит NULL, будут включены в результирующее множество. То есть запрос извлечет всех сотрудников, для которых столбец `manager_emp_id` содержит NULL или не NULL и содержит значение, отличное от 11.

## Куда идем дальше?

В этой главе речь шла о значении инструкции `WHERE` для различных типов операторов SQL, а также о многочисленных компонентах, используемых для построения этой инструкции. Учитывая всю важность инструкции `WHERE`, невозможно представить всю информацию в одной главе. Дополнительные сведения об инструкции `WHERE` можно найти:

- в главе 3, где подробно изучаются разнообразные условия объединения;
- в главе 5, в которой исследуются различные типы подзапросов и соответствующие операторы для вычисления их результатов;
- в главе 6, где описываются многочисленные методы обработки значений даты и времени;
- в главе 14, в которой некоторые аспекты инструкции `WHERE` рассматриваются с точки зрения производительности и эффективности.

Кроме того, дадим несколько советов, которые помогут при создании многих инструкций `WHERE`:

1. Тщательно проверяйте условия объединения. Убедитесь в том, что каждый набор данных в инструкции `FROM` правильно включен в объединение. Помните, что для некоторых объединений необходимо несколько условий (см. главу 3).
2. Избегайте ненужных объединений. То, что два набора данных инструкции `FROM` содержат один и тот же столбец, не обязательно влечет за собой добавление условия объединения в инструкцию `WHERE`. Выдает так, что избыточные данные копируются в различных таблицах в процессе денормализации. Посвятите немного времени знакомству со структурой базы данных и расспросите администратора базы данных о текущей модели данных.
3. Используйте скобки. Oracle поддерживает и приоритет операторов, и приоритет условий, то есть существует набор правил, определяю-



щих порядок вычисления. Но все же наиболее надежным как для вас, так и для человека, который впоследствии будет сопровождать вашу программу, является использование скобок для явного указания порядка вычисления. Использование скобок для операторов, например  $(5 * p.inventory\_qty) + 2$  вместо  $5 * p.inventory\_qty + 2$ , делает очевидным порядок выполнения действий. Что касается условий, необходимо применять скобки всегда, когда используется оператор OR.

4. Используйте структурирование текста. Например, если в предыдущей строке есть открывающая скобка и нет соответствующей закрывающей, сделайте в новой строке отступ, чтобы показать, что она является продолжением предыдущей.
5. При использовании OR ставьте первым то условие, вычисление которого требует наименьших усилий. Если первое условие будет оценено как TRUE, Oracle не будет заниматься вычислением остальных условий, что в некоторых случаях может означать существенную экономию времени. Такой подход полезен при работе со связанными запросами, которые обычно выполняются по одному разу для каждой строки.
6. Должным образом обрабатывайте NULL. Написав инструкцию WHERE, проверьте, как каждое из условий обрабатывает значения NULL. Не пожалейте времени на изучение определений таблиц вашей базы данных, чтобы точно знать, какие столбцы допускают использование NULL.
7. Пополните свою библиотеку книгами по началам логики и теории множеств. Хотя понимание этих двух вопросов не обязательно обеспечит вам приглашение на большее количество вечеринок, оно без сомнения повысит вашу квалификацию SQL-программиста.

# 3

## Объединения

Самодостаточных вещей практически не бывает. Нет такого магазина, в котором удалось бы купить все, что нужно. В таблицах базы данных все аналогично. Часто бывает необходима информация из нескольких таблиц. Конструкция языка SQL, комбинирующая данные двух и более таблиц, называется *объединением (join)*. В данной главе будут рассмотрены объединения, их типы и способы использования.

Объединение – это SQL-запрос, который извлекает информацию из двух или более таблиц или представлений. При указании в инструкции FROM нескольких таблиц или представлений Oracle выполняет объединение, связывая вместе строки различных таблиц. Существует несколько типов объединений:

### *Внутренние объединения (inner joins)*

Внутренние объединения – это стандартный вариант объединения, который возвращает строки, удовлетворяющие условию объединения. Каждая строка, возвращенная внутренним объединением, содержит данные всех таблиц, включенных в объединение.

### *Внешние объединения (outer join)*

Внешние объединения – это расширение внутренних. Внешнее объединение возвращает строки, удовлетворяющие условию объединения, а также те строки одной таблицы, для которых не найдено строк другой таблицы, отвечающих условию объединения.

### *Самообъединения (self joins)*

Самообъединение – это объединение таблицы с ней самой.

В последующих разделах каждый из типов объединений будет подробно рассмотрен на примерах.

# Внутренние объединения

Внутреннее объединение возвращает строки, удовлетворяющие условию объединения. Давайте рассмотрим понятие «объединение» на примере. Пусть необходимо вывести фамилию и название подразделения для каждого сотрудника. Используем следующий оператор SQL:

```
SELECT E.LNAME, D.NAME  
FROM EMPLOYEE E, DEPARTMENT D  
WHERE E.DEPT_ID = D.DEPT_ID;
```

LNAME	NAME
SMITH	RESEARCH
ALLEN	SALES
WARD	SALES
JONES	RESEARCH
MARTIN	SALES
BLAKE	SALES
CLARK	ACCOUNTING
SCOTT	RESEARCH
KING	ACCOUNTING
TURNER	SALES
ADAMS	RESEARCH
JAMES	SALES
FORD	RESEARCH
MILLER	ACCOUNTING

14 rows selected.

В этом примере запрос обращается к двум таблицам, так как фамилия служащего хранится в таблице EMPLOYEE, а название подразделения – в таблице DEPARTMENT. Обратите внимание на то, что в инструкции FROM названия двух таблиц, EMPLOYEE и DEPARTMENT, перечислены через запятую. Если нужно объединить три и более таблиц, укажите все таблицы в инструкции FROM, перечислив их через запятую. В списке оператора SELECT могут упоминаться столбцы из любой таблицы, указанной в инструкции FROM.

Обратите внимание на использование псевдонимов таблиц. При выборке данных из нескольких таблиц использование псевдонимов для названий таблиц – это установившаяся практика. При наличии неоднозначности названий столбцов использование псевдонимов (или названий) таблиц необходимо для недвусмысленного трактования названий. Например, столбец с названием DEPT\_ID присутствует в обеих таблицах. Поэтому в инструкции WHERE использованы псевдонимы E и D для установления равенства между столбцом DEPT\_ID таблицы EMPLOYEE и столбцом DEPT\_ID таблицы DEPARTMENT. Заметьте, что псевдонимы таблиц присутствуют и в инструкции SELECT, несмотря на однозначность названий. Заведите привычку использовать псевдонимы таблиц во всех вхождениях столбцов в запрос.



## Декартово произведение

Если при объединении двух таблиц условие объединения не указано, Oracle объединяет каждую строку первой таблицы с каждой строкой второй таблицы. Такое результирующее множество называется прямым, или декартовым, произведением. Количество строк декартова произведения равно произведению количеств строк в таблицах. Рассмотрим пример декартова произведения:

```
SELECT E.LNAME, D.NAME  
FROM EMPLOYEE E, DEPARTMENT D;
```

LNAME	NAME
SMITH	ACCOUNTING
ALLEN	ACCOUNTING
WARD	ACCOUNTING
JONES	ACCOUNTING
MARTIN	ACCOUNTING
BLAKE	ACCOUNTING

...

...

SCOTT	OPERATIONS
KING	OPERATIONS
TURNER	OPERATIONS
ADAMS	OPERATIONS
JAMES	OPERATIONS
FORD	OPERATIONS
MILLER	OPERATIONS

182 rows selected.

Условие объединения в запросе не определено, поэтому каждая строка таблицы EMPLOYEE объединена с каждой строкой таблицы DEPARTMENT. Очевидно, что в таком результирующем множестве мало пользы. В большинстве случаев в результирующем множестве, образованном как декартово произведение, содержатся вводящие в заблуждение строки. Поэтому если по какой-то причине вам не нужно именно декартово произведение, не забывайте об условии объединения для инструкции FROM, в которой указано несколько таблиц.

## Условие объединения

Обычно при выполнении объединения в инструкцию WHERE включается условие, которое устанавливает соответствие таблиц, указанных в инструкции FROM. Такое условие называется условием объединения. Условие объединения определяет, как следует объединять строки одной таблицы со строками другой. Как правило, условие объединения применяется к столбцам, которые являются внешними ключами таблиц.

В первом примере предыдущего раздела в инструкции WHERE было задано условие объединения, в котором указывалось равенство столбцов DEPT\_ID таблицы EMPLOYEE и таблицы DEPARTMENT:

```
WHERE E.DEPT_ID = D.DEPT_ID
```

Для выполнения объединения Oracle берет одну комбинацию строк из двух таблиц и проверяет истинность условия объединения. Если условие объединения истинно, Oracle включает данную комбинацию строк в результирующее множество. Процесс повторяется для всех сочетаний строк двух таблиц. Приведем несколько важных фактов, касающихся условия объединения.

- Нет необходимости включать столбцы, входящие в условие объединения, в список SELECT. В следующем примере условие объединения содержит столбец DEPT\_ID таблицы EMPLOYEE и таблицы DEPARTMENT, но при этом столбец DEPT\_ID не участвует в выборке:

```
SELECT E.LNAME, D.NAME  
FROM EMPLOYEE E, DEPARTMENT D  
WHERE E.DEPT_ID = D.DEPT_ID;
```

- Обычно условие объединения указывается для столбцов, являющихся внешним ключом одной таблицы и первичным или уникальным ключом другой таблицы. Однако можно использовать и другие столбцы. Каждое условие объединения затрагивает столбцы, которые устанавливают связь между двумя таблицами.
- Условие объединения может включать в себя несколько столбцов. Так обычно бывает, если внешний ключ состоит из нескольких столбцов.
- Общее количество условий объединения всегда на единицу меньше общего количества таблиц.
- Условия объединения должны содержать столбцы с совместимыми типами данных. Обратите внимание на то, что типы данных должны быть *совместимыми*, но не обязаны *совпадать*. При необходимости Oracle выполняет автоматическое преобразование типа.
- Оператор равенства (=) не обязательно должен входить в условие объединения. Возможно использование других операторов. В объединениях могут участвовать операторы, о которых будет рассказано далее в этом разделе.

## Объединения по равенству и объединения не по равенству

Условие объединения определяет, к какому типу относится объединение: объединения по равенству или объединения не по равенству. Если условие объединения связывает таблицы, ставя знак равенства между столбцами таблиц, такое объединение называется *объединением по ра-*



венству (*equi-join*). Если же условие объединения связывает таблицы, используя оператор, отличный от оператора равенства, то объединение называется *объединением не по равенству (non-equi-join)*. Запрос может содержать как одни, так и другие объединения.

Объединения по равенству распространены более широко. Например, если вы хотите перечислить все детали всех поставщиков, то объединяете таблицу SUPPLIER с таблицей PART, приравняв столбцы SUPPLIER\_ID двух таблиц:

```
SELECT S.NAME SUPPLIER_NAME, P.NAME PART_NAME
FROM SUPPLIER S, PART P
WHERE S.SUPPLIER_ID = P.SUPPLIER_ID;
```

Но бывают случаи, когда для получения необходимой информации требуется объединение не по равенству. Например, если нужно вывести INVENTORY\_CLASS для каждой детали из таблицы PART, выполняем такой запрос:

```
SELECT P.NAME PART_NAME, C.INV_CLASS INV_CLASS
FROM PART P, INVENTORY_CLASS C
WHERE P.UNIT_COST BETWEEN C.LOW_COST AND C.HIGH_COST;
```

Для связи столбца UNIT\_COST таблицы PART со столбцами LOW\_COST и HIGH\_COST таблицы INVENTORY\_CLASS использован оператор BETWEEN.

## Внешние объединения

При объединении двух таблиц может возникнуть необходимость вывести все строки одной из таблиц, даже если для них не существует соответствующих строк во второй таблице. Рассмотрим две таблицы: поставщиков (SUPPLIER) и деталей (PART):

```
SELECT * FROM SUPPLIER;
```

```
SUPPLIER_ID NAME
```

```
-----
101 Pacific Disks, Inc.
102 Silicon Valley MicroChips
103 Blue River Electronics
```

```
SELECT * FROM PART;
```

```
PART_NBR NAME          SUPPLIER_ID STATUS INVENTORY_QTY UNIT_COST RESUPPLY_DATE
-----
HC211    20 GB Hard Disk      101 ACTIVE          5      2000 12-DEC-00
P3000    3000 MHz Processor    102 ACTIVE         12        600 03-NOV-00
```

Если нужно вывести всех поставщиков и поставляемые ими детали, естественно использовать следующий запрос:

```
SELECT S.SUPPLIER_ID, S.NAME SUPPLIER_NAME, P.PART_NBR, P.NAME PART_NAME
FROM SUPPLIER S, PART P
```

WHERE S.SUPPLIER\_ID = P.SUPPLIER\_ID;

SUPPLIER_ID	SUPPLIER_NAME	PART_NBR	PART_NAME
101	Pacific Disks, Inc.	HD211	20 GB Hard Disk
102	Silicon Valley MicroChips	P3000	3000 MHz Processor

Обратите внимание на то, что, хотя поставщиков трое, запрос выводит только двоих, потому что третий поставщик (Blue River Electronics) в данный момент ничего не поставляет. Когда Oracle выполняет объединение между таблицами SUPPLIER и PART, сопоставляются столбцы SUPPLIER\_ID этих двух таблиц (как указано в условии объединения). Так как для SUPPLIER\_ID = 103 не существует соответствующих записей в таблице PART, этот поставщик не включается в результирующее множество. Такой тип объединения является наиболее естественным и называется *внутренним объединением*.



Понятие внутреннего объединения легче пояснить в терминах декартова произведения. При выполнении объединения таблиц SUPPLIER и PART сначала формируется декартово произведение (физически оно не материализуется), а затем условия инструкции WHERE ограничивают результат только теми строками, в которых совпадают значения SUPPLIER\_ID.

Но хотелось бы получить полный список поставщиков, включающий и тех, кто в данный момент ничего не поставляет. Oracle предоставляет специальный тип объединения, который позволяет включать в результирующее множество строки одной таблицы, для которых не найдены соответствующие строки в другой таблице. Такое объединение называется *внешним (outer)*. Внешнее объединение позволит вывести строки для всех поставщиков, а если поставщик в настоящий момент поставляет какие-то детали, то и соответствующие строки деталей. Если в настоящий момент поставщик не поставляет детали, в результирующем множестве в столбцах таблицы PART будут возвращены значения NULL.

Синтаксис внешнего объединения несколько отличается от синтаксиса внутреннего объединения. Применяется специальный оператор, называемый *оператором внешнего объединения*, который выглядит как знак «плюс», заключенный в круглые скобки, то есть (+). Этот оператор используется в условии объединения инструкции WHERE вслед за именем поля той таблицы, которую вы хотите рассматривать как необязательную. В рассматриваемом примере про детали и поставщиков таблица PART не содержит информацию об одном поставщике. Просто добавляем оператор (+) к условию объединения со стороны таблицы PART. Запрос и результирующее множество будут выглядеть следующим образом:

```
SELECT S.SUPPLIER_ID, S.NAME SUPPLIER_NAME, P.PART_NBR, P.NAME PART_NAME  
FROM SUPPLIER S, PART P
```



WHERE S.SUPPLIER\_ID = P.SUPPLIER\_ID (+);

SUPPLIER_ID	SUPPLIER_NAME	PART_NBR	PART_NAME
101	Pacific Disks, Inc.	HD211	20 GB Hard Disk
102	Silicon Valley MicroChips	P3000	3000 MHz Processor
103	Blue River Electronics		

Заметьте, что оператор (+) следует за P.SUPPLIER\_ID, что делает таблицу PART необязательной (в данном объединении). Если поставщик ничего не поставляет в настоящий момент, Oracle создаст для данного поставщика запись в таблице PART со значениями NULL во всех ячейках. Результирующее множество теперь содержит всех поставщиков независимо от состояния их текущих поставок. Как видите, столбцы PART для поставщика с идентификатором 103 содержат NULL.

Оператор внешнего объединения (+) может появляться как в левой, так и в правой части условия объединения. Вы только должны быть уверены в том, что применяете оператор к соответствующей таблице (в контексте данного запроса). Например, если поменять местами части оператора равенства из предыдущего примера, это никак не повлияет на результат:

```
SELECT S.SUPPLIER_ID, S.NAME SUPPLIER_NAME, P.PART_NBR, P.NAME PART_NAME
FROM SUPPLIER S, PART P
WHERE P.SUPPLIER_ID (+) = S.SUPPLIER_ID;
```

SUPPLIER_ID	SUPPLIER_NAME	PART_NBR	PART_NAME
101	Pacific Disks, Inc.	HD211	20 GB Hard Disk
102	Silicon Valley MicroChips	P3000	3000 MHz Processor
103	Blue River Electronics		

Но если сопоставить оператор (+) не той таблице, которой нужно, можно получить неожиданный результат. Например:

```
SELECT S.SUPPLIER_ID, S.NAME SUPPLIER_NAME, P.PART_NBR, P.NAME PART_NAME
FROM SUPPLIER S, PART P
WHERE P.SUPPLIER_ID = S.SUPPLIER_ID (+);
```

SUPPLIER_ID	SUPPLIER_NAME	PART_NBR	PART_NAME
101	Pacific Disks, Inc.	HD211	20 GB Hard Disk
102	Silicon Valley MicroChips	P3000	3000 MHz Processor

В данном случае оператор внешнего объединения находится в условии объединения со стороны таблицы SUPPLIER. Тем самым вы просите Oracle вывести детали и их поставщиков, а также те детали, которым не соответствует никакой поставщик. Но в рассматриваемой базе данных всем деталям сопоставлен поставщик. Поэтому результат аналогичен результату выполнения внутреннего объединения.

## Ограничения, налагаемые на внешние объединения

Существует ряд правил и ограничений, относящихся к использованию внешних объединений в запросах. Если в запросе выполняется внешнее объединение, Oracle не разрешает использовать в этом же запросе некоторые другие операции. Далее мы поговорим о таких ограничениях и о некоторых способах их обхода.

- Оператор внешнего объединения может присутствовать только в одной части условия объединения. При попытке использовать его в обеих частях возникает ошибка ORA-1468. Например:

```
SELECT S.SUPPLIER_ID, S.NAME SUPPLIER_NAME, P.PART_NBR, P.NAME PART_NAME
FROM SUPPLIER S, PART P
WHERE S.SUPPLIER_ID (+) = P.SUPPLIER_ID (+);
WHERE S.SUPPLIER_ID (+) = P.SUPPLIER_ID (+)
```

ERROR at line 3:

ORA-01468: a predicate may reference only one outer-joined table



Если вы хотите создать двустороннее внешнее объединение, используя оператор (+) в обеих частях условия объединения, вам будет полезен следующий раздел – «Полные внешние объединения».

- Если в объединении участвует более двух таблиц, то каждая из таблиц в запросе не может участвовать во внешнем объединении с более чем одной другой таблицей. Давайте рассмотрим пример:

DESC EMPLOYEE

Name	Null?	Type
EMP_ID	NOT NULL	NUMBER(5)
FNAME		VARCHAR2(20)
LNAME		VARCHAR2(20)
DEPT_ID	NOT NULL	NUMBER(5)
MANAGER_EMP_ID		NUMBER(5)
SALARY		NUMBER(5)
HIRE_DATE		DATE
JOB_ID		NUMBER(3)

DESC JOB

Name	Null?	Type
JOB_ID	NOT NULL	NUMBER(3)
FUNCTION		VARCHAR2(30)

DESC DEPARTMENT

Name	Null?	Type
DEPT_ID	NOT NULL	NUMBER(5)
NAME		VARCHAR2(20)
LOCATION_ID		NUMBER(3)



Если необходимо вывести для всех сотрудников должность и название подразделения и при этом включить в результирующее множество все подразделения и должности, которым в настоящий момент не сопоставлены никакие сотрудники, вы, вероятно, попытаетесь объединить таблицу EMPLOYEE с таблицей JOB и таблицей DEPARTMENT. Оба объединения будут внешними. Но так как одна таблица не может быть внешне объединена с несколькими таблицами, то будет выдана следующая ошибка:

```
SELECT E.LNAME, J.FUNCTION, D.NAME
FROM EMPLOYEE E, JOB J, DEPARTMENT D
WHERE E.JOB_ID (+) = J.JOB_ID
AND E.DEPT_ID (+) = D.DEPT_ID;

WHERE E.JOB_ID (+) = J.JOB_ID
      *
```

ERROR at line 3:  
ORA-01417: a table may be outer joined to at most one other table

Выходом может стать создание представления с внешним объединением двух таблиц, а затем выполнение внешнего объединения этого представления с третьей таблицей:

```
CREATE VIEW V_EMP_JOB
AS SELECT E.DEPT_ID, E.LNAME, J.FUNCTION
FROM EMPLOYEE E, JOB J
WHERE E.JOB_ID (+) = J.JOB_ID;

SELECT V.LNAME, V.FUNCTION, D.NAME
FROM V_EMP_JOB V, DEPARTMENT D
WHERE V.DEPT_ID (+) = D.DEPT_ID;
```

Вместо того чтобы создавать представление, можно получить тот же результат, используя встроенное представление:

```
SELECT V.LNAME, V.FUNCTION, D.NAME
FROM (SELECT E.DEPT_ID, E.LNAME, J.FUNCTION
      FROM EMPLOYEE E, JOB J
      WHERE E.JOB_ID (+) = J.JOB_ID) V, DEPARTMENT D
WHERE V.DEPT_ID (+) = D.DEPT_ID;
```

Встроенные представления рассматриваются в главе 5.

- В условии внешнего объединения, содержащем оператор (+), запрещено использование оператора IN. Например:

```
SELECT E.LNAME, J.FUNCTION
FROM EMPLOYEE E, JOB J
WHERE E.JOB_ID (+) IN (668, 670, 667);
WHERE E.JOB_ID (+) IN (668, 670, 667)
      *
```

ERROR at line 3:  
ORA-01719: outer join operator (+) not allowed in operand of OR or IN

- Условие внешнего объединения, содержащее оператор (+), нельзя комбинировать с другими условиями при помощи оператора OR. Например:

```
SELECT E.LNAME, D.NAME
FROM EMPLOYEE E, DEPARTMENT D
WHERE E.DEPT_ID = D.DEPT_ID (+)
OR D.DEPT_ID = 10;
WHERE E.DEPT_ID = D.DEPT_ID (+)
```

ERROR at line 3:

ORA-01719: outer join operator (+) not allowed in operand of OR or IN

- Условие внешнего объединения, содержащее оператор (+), не может содержать подзапрос. Например:

```
SELECT E.LNAME
FROM EMPLOYEE E
WHERE E.DEPT_ID (+) =
(SELECT DEPT_ID FROM DEPARTMENT WHERE NAME = 'ACCOUNTING');
(SELECT DEPT_ID FROM DEPARTMENT WHERE NAME = 'ACCOUNTING')
```

ERROR at line 4:

ORA-01798: a column may not be outer-joined to a subquery

Чтобы достичь желаемого эффекта и избежать ошибки, можно использовать встроенное представление:

```
SELECT E.LNAME
FROM EMPLOYEE E,
(SELECT DEPT_ID FROM DEPARTMENT WHERE NAME = 'ACCOUNTING') V
WHERE E.DEPT_ID (+) = V.DEPT_ID;
```

Встроенные представления обсуждаются в главе 5.

## Полные внешние объединения

Внешнее объединение расширяет результат внутреннего объединения, включая в него строки одной таблицы (например, таблицы А), для которых не найдено соответствующих строк в другой таблице (например, таблице В). Важно понимать, что внешнее объединение не включает в результирующее множество те строки таблицы В, которым не соответствуют строки таблицы А. Другими словами, внешнее объединение является однонаправленным. Возможны случаи, в которых хотелось бы использовать двунаправленное внешнее объединение, то есть включить в результат все строки А и В, которые:

- Входят в результат внутреннего объединения.
- Строки таблицы А, которым не соответствуют строки таблицы В.
- Строки таблицы В, которым не соответствуют строки таблицы А.



Давайте рассмотрим пример. Обратимся к двум таблицам: LOCATION и DEPARTMENT:

DESC LOCATION		
Name	Null?	Type
-----		
LOCATION_ID	NOT NULL	NUMBER(3)
REGIONAL_GROUP		VARCHAR2(20)
DESC DEPARTMENT		
Name	Null?	Type
-----		
DEPT_ID	NOT NULL	NUMBER(5)
NAME		VARCHAR2(20)
LOCATION_ID		NUMBER(3)

Предположим, что в таблице LOCATION существуют такие записи о местоположении, которым не соответствуют никакие подразделения из таблицы DEPARTMENT, и что, в то же время, в таблице DEPARTMENT существуют подразделения без значения LOCATION\_ID, указывающего на их местоположение. Если выполнить внутреннее объединение двух этих таблиц, будут выведены только подразделения и местоположения, которым соответствуют строки обеих таблиц:

```
SELECT D.DEPT_ID, D.NAME, L.REGIONAL_GROUP
FROM DEPARTMENT D, LOCATION L
WHERE D.LOCATION_ID = L.LOCATION_ID;
```

DEPT_ID	NAME	REGIONAL_GROUP
-----		
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
12	RESEARCH	NEW YORK
13	SALES	NEW YORK
14	OPERATIONS	NEW YORK
23	SALES	DALLAS
24	OPERATIONS	DALLAS
34	OPERATIONS	CHICAGO
43	SALES	BOSTON

11 rows selected.

По некоторым адресам не существует ни одного подразделения. Чтобы включить эти местоположения в список вывода, необходимо выполнить внешнее объединение, применив оператор (+) со стороны подразделения. Тем самым вы сделаете таблицу DEPARTMENT необязательной в данном запросе. Заметьте, что на место отсутствующих в таблице DEPARTMENT данных Oracle подставляет NULL:

```
SELECT D.DEPT_ID, D.NAME, L.REGIONAL_GROUP
FROM DEPARTMENT D, LOCATION L
```

WHERE D.LOCATION\_ID (+) = L.LOCATION\_ID;

DEPT_ID	NAME	REGIONAL_GROUP
-----		
10	ACCOUNTING	NEW YORK
12	RESEARCH	NEW YORK
14	OPERATIONS	NEW YORK
13	SALES	NEW YORK
30	SALES	CHICAGO
34	OPERATIONS	CHICAGO
20	RESEARCH	DALLAS
23	SALES	DALLAS
24	OPERATIONS	DALLAS
		SAN FRANCISCO
40	OPERATIONS	BOSTON
43	SALES	BOSTON

12 rows selected.

Существуют и подразделения, которым не соответствует ни одно из местоположений. Если вы хотите включить эти подразделения в результирующее множество, выполните внешнее объединение, применив оператор (+) со стороны таблицы LOCATION:

```
SELECT D.DEPT_ID, D.NAME, L.REGIONAL_GROUP
FROM DEPARTMENT D, LOCATION L
WHERE D.LOCATION_ID = L.LOCATION_ID (+) ;
```

DEPT_ID	NAME	REGIONAL_GROUP
-----		
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
12	RESEARCH	NEW YORK
13	SALES	NEW YORK
14	OPERATIONS	NEW YORK
23	SALES	DALLAS
24	OPERATIONS	DALLAS
34	OPERATIONS	CHICAGO
43	SALES	BOSTON
50	MARKETING	
60	CONSULTING	

13 rows selected.

Но из результата пропали строки с местоположениями, которым не было сопоставлено ни одно подразделение. Чтобы включить в результирующее множество и подразделения без местоположения, и местоположения без подразделений, вы, вероятно, попытаетесь использовать двунаправленное внешнее объединение (правильный термин – *полное внешнее объединение* (*full outer join*)), подобное приведенному ниже:

```
SELECT D.DEPT_ID, D.NAME, L.REGIONAL_GROUP
```



```
FROM DEPARTMENT D, LOCATION L
WHERE D.LOCATION_ID (+) = L.LOCATION_ID (+);
WHERE D.LOCATION_ID (+) = L.LOCATION_ID (+)
```

ERROR at line 3:

ORA-01468: a predicate may reference only one outer-joined table

Как видите, двунаправленное внешнее объединение не разрешено. Выходом становится слияние (UNION) двух операторов SELECT. В следующем примере первый оператор SELECT представляет внешнее объединение, в котором необязательной таблицей является DEPARTMENT. Во втором операторе SELECT необязательной таблицей является LOCATION. Выполнив слияние двух запросов, вы получаете все подразделения и все местоположения. Операция UNION уничтожает дублирующиеся строки, и в результате получается полное внешнее объединение:

```
SELECT D.DEPT_ID, D.NAME, L.REGIONAL_GROUP
FROM DEPARTMENT D, LOCATION L
WHERE D.LOCATION_ID (+) = L.LOCATION_ID
UNION
SELECT D.DEPT_ID, D.NAME, L.REGIONAL_GROUP
FROM DEPARTMENT D, LOCATION L
WHERE D.LOCATION_ID = L.LOCATION_ID (+) ;
```

DEPT_ID	NAME	REGIONAL_GROUP
10	ACCOUNTING	NEW YORK
12	RESEARCH	NEW YORK
13	SALES	NEW YORK
14	OPERATIONS	NEW YORK
20	RESEARCH	DALLAS
23	SALES	DALLAS
24	OPERATIONS	DALLAS
30	SALES	CHICAGO
34	OPERATIONS	CHICAGO
40	OPERATIONS	BOSTON
43	SALES	BOSTON
50	MARKETING	
60	CONSULTING	
		SAN FRANCISCO

14 rows selected.

Как видите, этот UNION-запрос выдает все строки, которые хотелось увидеть в полном внешнем объединении. О запросах UNION подробно рассказано в главе 7.



Oracle9i представляет новый ANSI-совместимый синтаксис объединения, который разрешает использование полных внешних объединений более простым способом. Об этом новом синтаксисе мы поговорим в конце главы.

## Самообъединения

Выпадают случаи, когда одна строка таблицы связана с другой строкой той же самой таблицы. Рассмотрим, например, таблицу EMPLOYEE. Руководитель каждого служащего также является служащим. Строки, соответствующие им обоим, находятся в одной таблице – EMPLOYEE. Отношение между этими строками задается в столбце MANAGER\_EMP\_ID:

```
CREATE TABLE EMPLOYEE (  
EMP_ID          NUMBER (5) NOT NULL PRIMARY KEY,  
FNAME          VARCHAR2 (20),  
LNAME          VARCHAR2 (20),  
DEPT_ID        NUMBER (5),  
MANAGER_EMP_ID NUMBER (5) REFERENCES EMPLOYEE(EMP_ID),  
SALARY         NUMBER (5),  
HIRE_DATE      DATE,  
JOB_ID         NUMBER (3));
```

Чтобы получить информацию о служащем и его руководителе, необходимо объединить таблицу EMPLOYEE с ней самой. Для этого нужно дважды указать ее в инструкции FROM, используя два разных псевдонима, чтобы две таблицы EMPLOYEE воспринимались как разные. В следующем примере выводится список имен служащих и их руководителей:

```
SELECT E.LNAME EMPLOYEE, M.LNAME MANAGER  
FROM EMPLOYEE E, EMPLOYEE M  
WHERE E.MANAGER_EMP_ID = M.EMP_ID;
```

EMPLOYEE	MANAGER
SMITH	FORD
ALLEN	BLAKE
WARD	BLAKE
JONES	KING
MARTIN	BLAKE
BLAKE	KING
CLARK	KING
SCOTT	JONES
TURNER	BLAKE
ADAMS	SCOTT
JAMES	BLAKE
FORD	JONES
MILLER	CLARK

13 rows selected.

Обратите внимание на то, что таблица EMPLOYEE дважды входит в инструкцию FROM с двумя разными псевдонимами. Заметьте, что условие объединения читается как «когда MANAGER\_EMP\_ID служащего равен EMP\_ID его руководителя».

## Внешние самообъединения

В таблице EMPLOYEE 14 строк, но предыдущий запрос вернул только 13. Дело в том, что существует служащий без MANAGER\_EMP\_ID. При выполнении внутреннего самообъединения Oracle исключает эту строку из результирующего множества. Чтобы включить в результат служащего без MANAGER\_EMP\_ID, необходимо внешнее объединение:

```
SELECT E.LNAME EMPLOYEE, M.LNAME MANAGER
FROM EMPLOYEE E, EMPLOYEE M
WHERE E.MANAGER_EMP_ID = M.EMP_ID (+);
```

EMPLOYEE	MANAGER
SMITH	FORD
ALLEN	BLAKE
WARD	BLAKE
JONES	KING
MARTIN	BLAKE
BLAKE	KING
CLARK	KING
SCOTT	JONES
KING	
TURNER	BLAKE
ADAMS	SCOTT
JAMES	BLAKE
FORD	JONES
MILLER	CLARK

14 rows selected.

Будьте внимательны, вставляя в условие объединения оператор (+). Если вы поставите (+) не с той стороны, то получите бессмысленный и непонятный результат. В нашем случае необходимо сделать обязательной ту таблицу EMPLOYEE, из которой выбираются фамилии руководителей.

## Самообъединения не по равенству

В предыдущем примере использовалось самообъединение по равенству. Но бывают ситуации, когда необходимо выполнить самообъединение не по равенству. Снова обратимся к примеру. Предположим, что вам поручено организовать турнир по баскетболу между подразделениями компании. В вашу задачу входит создание команд и составление графика матчей. Вы обращаетесь к таблице подразделений DEPARTMENT и получаете следующий результат:

```
SELECT NAME FROM DEPARTMENT;
```

NAME
ACCOUNTING



RESEARCH  
SALES  
OPERATIONS

В компании четыре подразделения, и чтобы соревнование было честным, вы решаете, что каждое подразделение по одному разу сыграет с остальными тремя, и то подразделение, которое одержит больше побед, и станет победителем. Вы недавно были на курсах повышения квалификации по Oracle SQL и знаете, как использовать самообъединение, поэтому выполняете такой запрос:

```
SELECT D1.NAME TEAM1, D2.NAME TEAM2  
FROM DEPARTMENT D1, DEPARTMENT D2;
```

TEAM1	TEAM2
ACCOUNTING	ACCOUNTING
RESEARCH	ACCOUNTING
SALES	ACCOUNTING
OPERATIONS	ACCOUNTING
ACCOUNTING	RESEARCH
RESEARCH	RESEARCH
SALES	RESEARCH
OPERATIONS	RESEARCH
ACCOUNTING	SALES
RESEARCH	SALES
SALES	SALES
OPERATIONS	SALES
ACCOUNTING	OPERATIONS
RESEARCH	OPERATIONS
SALES	OPERATIONS
OPERATIONS	OPERATIONS

16 rows selected.

Печальные результаты. Из университетского курса математики вы помните, что если каждая из четырех команд по разу сыграет с остальными, комбинаций получится шесть. Но SQL-запрос возвращает 16 строк. Вы понимаете, что не указали условие объединения, поэтому в результате было выведено декартово произведение. Вставляете условие объединения, и теперь результирующее множество выглядит так:

```
SELECT D1.NAME TEAM1, D2.NAME TEAM2  
FROM DEPARTMENT D1, DEPARTMENT D2  
WHERE D1.DEPT_ID = D2.DEPT_ID;
```

TEAM1	TEAM2
ACCOUNTING	ACCOUNTING
RESEARCH	RESEARCH
SALES	SALES
OPERATIONS	OPERATIONS

Да... это совсем не то, что хотелось бы. Команда не может играть против себя самой. Вы осознаете ошибку, и тут возникает идея использовать объединение не по равенству. Вы переписываете запрос, применяя объединение не по равенству. Вы не хотите, чтобы команда играла сама против себя, и заменяете в условии объединения оператор «=» на «!=». Давайте посмотрим, что получилось:

```
SELECT D1.NAME TEAM1, D2.NAME TEAM2
FROM DEPARTMENT D1, DEPARTMENT D2
WHERE D1.DEPT_ID != D2.DEPT_ID;
```

TEAM1	TEAM2
RESEARCH	ACCOUNTING
SALES	ACCOUNTING
OPERATIONS	ACCOUNTING
ACCOUNTING	RESEARCH
SALES	RESEARCH
OPERATIONS	RESEARCH
ACCOUNTING	SALES
RESEARCH	SALES
OPERATIONS	SALES
ACCOUNTING	OPERATIONS
RESEARCH	OPERATIONS
SALES	OPERATIONS

12 rows selected.

Все равно ничего хорошего. В это результирующее множество входят перестановки, такие как (RESEARCH, ACCOUNTING), (ACCOUNTING, RESEARCH) и т. д. Получается, что каждая команда сыграет с остальными дважды. Необходимо избавиться от этих дублирующих друг друга перестановок. Вы обдумываете, не использовать ли ключевое слово DISTINCT. Но в данном случае DISTINCT не поможет, так как с его точки зрения строки (RESEARCH, ACCOUNTING) и (ACCOUNTING, RESEARCH) различны, но ведь для вас это не так. Поразмыслив еще, вы решаете попробовать применить оператор неравенства, отличный от «!=», и останавливаетесь на операторе «меньше чем» (<). И вот результат:

```
SELECT D1.NAME TEAM1, D2.NAME TEAM2
FROM DEPARTMENT D1, DEPARTMENT D2
WHERE D1.DEPT_ID < D2.DEPT_ID;
```

TEAM1	TEAM2
ACCOUNTING	RESEARCH
ACCOUNTING	SALES
RESEARCH	SALES
ACCOUNTING	OPERATIONS
RESEARCH	OPERATIONS
SALES	OPERATIONS

6 rows selected.

Получилось! Теперь у вас шесть комбинаций, каждая команда по одному разу играет со всеми остальными командами. Давайте посмотрим, почему эта версия запроса работает так, как надо. При выполнении данного запроса Oracle сначала генерирует декартово произведение из 16 строк. Затем оператор «меньше чем» (<) в условии объединения ограничивает результирующее множество теми строками, в которых DEPT\_ID команды 1 меньше, чем DEPT\_ID команды 2. Оператор «меньше чем» (<) устраняет повторения, так как указанному условию удовлетворяет только одна пара из каждой перестановки. Используя оператор «больше чем» (>), вы получили бы тот же результат, только значения команд 1 и 2 поменялись бы местами:

```
SELECT D1.NAME TEAM1, D2.NAME TEAM2
FROM DEPARTMENT D1, DEPARTMENT D2
WHERE D1.DEPT_ID > D2.DEPT_ID;
```

TEAM1	TEAM2
RESEARCH	ACCOUNTING
SALES	ACCOUNTING
OPERATIONS	ACCOUNTING
SALES	RESEARCH
OPERATIONS	RESEARCH
OPERATIONS	SALES

6 rows selected.

Не впадайте в уныние от того долгого и мучительного процесса, который предшествовал получению результата. Иногда такая неприятная работа – это единственный способ получения простого результата. Так уж устроена жизнь. Теперь, когда у вас есть правильные комбинации команд, давайте сделаем шаг вперед и назначим дату каждого матча. Используем в качестве точки отсчета времени «завтрашний» день:

```
SELECT D1.NAME TEAM1, D2.NAME TEAM2, SYSDATE + ROWNUM MATCH_DATE
FROM DEPARTMENT D1, DEPARTMENT D2
WHERE D1.DEPT_ID < D2.DEPT_ID;
```

TEAM1	TEAM2	MATCH_DATE
ACCOUNTING	RESEARCH	30-APR-01
ACCOUNTING	SALES	01-MAY-01
RESEARCH	SALES	02-MAY-01
ACCOUNTING	OPERATIONS	03-MAY-01
RESEARCH	OPERATIONS	04-MAY-01
SALES	OPERATIONS	05-MAY-01

6 rows selected.

Пришло время публиковать полученные данные на корпоративном сайте вместе с правилами соревнований. И вы свободны!



## Объединения и подзапросы

В некоторых случаях объединения удобно использовать для переформулировки операторов SELECT, содержащих подзапросы. Рассмотрим задачу получения списка поставщиков тех деталей, которых на складе осталось менее десяти единиц. Можно написать следующий запрос:

```
SELECT supplier_id, name
FROM supplier s
WHERE EXISTS (SELECT *
              FROM part p
              WHERE p.inventory_qty < 10
                 AND p.supplier_id = s.supplier_id);
```

Подзапрос данного оператора SELECT – это связанный подзапрос, что означает, что он будет выполнен для каждой строки таблицы поставщиков. Если предположить, что у вас нет индексов на столбцы INVENTORY\_QTY и SUPPLIER\_ID таблицы PART, то такой запрос может привести к многочисленным просмотрам всей таблицы PART. Можно переформулировать запрос, используя объединение, например:

```
SELECT s.supplier_id, s.name
FROM supplier s, part p
WHERE p.supplier_id = s.supplier_id
      AND p.inventory_qty < 10;
```

Какой из запросов (с объединением или с подзапросом) окажется более эффективным, зависит от конкретной ситуации. Вероятно, стоит протестировать оба способа, чтобы посмотреть, какой из них требует наименьших затрат.

## Операторы DML и представление объединения

Представление объединения (join view) – это представление, созданное на основе объединения. Существует ряд ограничений, касающихся использования операторов DML (INSERT, UPDATE или DELETE) для представлений объединений. Всегда думайте о том, что происходит, когда вы вставляете строку в представление объединения – в какую именно таблицу она попадет? И что случится при удалении строки из представления объединения – из какой таблицы она будет удалена? В данном разделе вы получите ответы на эти вопросы.

Для того чтобы быть изменяемым, представление объединения не должно содержать ничего из нижеперечисленного:

- Иерархические инструкции запросов, такие как START WITH или CONNECT BY.
- Инструкции GROUP BY или HAVING.

- Операции с множествами, такие как UNION, UNION ALL, INTERSECT или MINUS.
- Обобщающие функции, такие как AVG, COUNT, MAX, MIN, SUM и т. д.
- Оператор DISTINCT.
- Псевдостолбец ROWNUM.

Оператор DML с представлением объединения может изменять только базовую таблицу представления. Для того чтобы представление объединения было изменяемым, кроме выполнения указанных правил также необходимо наличие одной таблицы, сохраняющей ключи.

## Таблицы, сохраняющие ключи

Таблица, сохраняющая ключи, – это наиболее важное из требований, которые должны быть выполнены для того, чтобы представление объединения было изменяемым. В объединении таблица называется *сохраняющей ключи (key-preserved table)*, если ее ключи сохраняются в процессе выполнения объединения: каждый ключ таблицы также является и ключом результирующего множества. Каждый первичный (уникальный) ключ базовой таблицы также должен быть уникальным для результирующего множества объединения. Давайте рассмотрим понятие таблицы, сохраняющей ключи, на примере:

### DESC EMPLOYEE

Name	Null?	Type
EMP_ID	NOT NULL	NUMBER(5)
LNAME		VARCHAR2(20)
FNAME		VARCHAR2(20)
DEPT_ID		NUMBER(5)
MANAGER_EMP_ID		NUMBER(5)
SALARY		NUMBER(5)
HIRE_DATE		DATE
JOB_ID		NUMBER(3)

### DESC RETAILER

Name	Null?	Type
RTLR_NBR	NOT NULL	NUMBER(6)
NAME		VARCHAR2(45)
ADDRESS		VARCHAR2(40)
CITY		VARCHAR2(30)
STATE		VARCHAR2(2)
ZIP_CODE		VARCHAR2(9)
AREA_CODE		NUMBER(3)
PHONE_NUMBER		NUMBER(7)
SALESPERSON_ID		NUMBER(4)
CREDIT_LIMIT		NUMBER(9, 2)
COMMENTS		LONG

```
CREATE OR REPLACE VIEW V_RTLR_EMP AS
SELECT C.RTLR_NBR, C.NAME, C.CITY, E.EMP_ID, C.SALESPERSON_ID, E.LNAME SALES_REP
FROM RETAILER C, EMPLOYEE E
WHERE C.SALESPERSON_ID = E.EMP_ID;
```

View created.

```
SELECT * FROM V_RTLR_EMP;
```

RTLР_NBR	NAME	CITY	EMP_ID	SALES_REP
100	JOCKSPORTS	BELMONT	7844	TURNER
101	TKB SPORT SHOP	REDWOOD CITY	7521	WARD
102	VOLLYRITE	BURLINGAME	7654	MARTIN
103	JUST TENNIS	BURLINGAME	7521	WARD
104	EVERY MOUNTAIN	CUPERTINO	7499	ALLEN
105	K + T SPORTS	SANTA CLARA	7844	TURNER
106	SHAPE UP	PALO ALTO	7521	WARD
107	WOMENS SPORTS	SUNNYVALE	7499	ALLEN
201	STADIUM SPORTS	NEW YORK	7499	ALLEN
202	HOOPS	LEICESTER	7499	ALLEN
203	REBOUND SPORTS	NEW YORK	7499	ALLEN
204	THE POWER FORWARD	DALLAS	7654	MARTIN
205	POINT GUARD	YONKERS	7499	ALLEN
206	THE COLISEUM	SCARSDALE	7499	ALLEN
207	FAST BREAK	CONCORD	7499	ALLEN
208	AL AND BOB'S SPORTS	AUSTIN	7654	MARTIN
211	AT BAT	BROOKLINE	7499	ALLEN
212	ALL SPORT	BROOKLYN	7844	TURNER
213	GOOD SPORT	SUNNYSIDE	7844	TURNER
214	AL'S PRO SHOP	SPRING	7654	MARTIN
215	BOB'S FAMILY SPORTS	HOUSTON	7654	MARTIN
216	THE ALL AMERICAN	CHELSEA	7499	ALLEN
217	HIT, THROW, AND RUN	GRAPEVINE	7854	MARTIN
218	THE OUTFIELD	FLUSHING	7499	ALLEN
221	WHEELS AND DEALS	HOUSTON	7844	TURNER
222	JUST BIKES	DALLAS	7844	TURNER
223	VELO SPORTS	MALDEN	7499	ALLEN
224	JOE'S BIKE SHOP	GRAND PRARIE	7844	TURNER
225	BOB'S SWIM, CYCLE, AND RUN	IRVING	7844	TURNER
226	CENTURY SHOP	HUNTINGTON	7521	WARD
227	THE TOUR	SOMERVILLE	7499	ALLEN
228	FITNESS FIRST	JACKSON HEIGHTS	7521	WARD

32 rows selected.

Представление V\_RTLR\_EMP – это объединение таблиц RETAILER и EMPLOYEE по столбцам RETAILER.SALESPERSON\_ID и EMPLOYEE.EMP\_ID. Есть ли в этом представлении таблица, сохраняющая ключи? Которая из них – или, может быть, обе? Если вы посмотрите на отношение между двумя таблицами и на условие объединения, то заметите, что RTLР\_NBR является ключом таблицы RETAILER, а также ключом результата объединения. Это обусловлено тем, что существует ровно одна строка таблицы RETAILER



для каждой строки `V_RTLR_EMP`, и каждая строка представления имеет уникальное значение `RTLNR_NBR`. Поэтому таблица `RETAILER` является сохраняющей ключи в данном представлении объединения. Что насчет таблицы `EMPLOYEE`? Ключ таблицы `EMPLOYEE` не сохраняется в процессе объединения, так как в представлении значение `EMP_ID` не является уникальным и, следовательно, не может быть ключом результирующего множества. Поэтому таблица `EMPLOYEE` не является сохраняющей ключи в рассматриваемом представлении объединения.

Приведем несколько важных фактов, относящихся к таблицам, сохраняющим ключи:

- Сохранение ключей – это свойство таблицы в пределах представления объединения, а не внутреннее свойство самой таблицы. Таблица может не сохранять ключи в одном представлении объединения и сохранять в другом. Например, если создать представление объединения, соединив таблицы `EMPLOYEE` и `DEPARTMENT` по столбцу `DEPT_ID`, то в результирующем представлении таблица `EMPLOYEE` будет сохраняющей ключи, а `DEPARTMENT` – нет.
- Ключевой столбец (столбцы) таблицы не обязаны присутствовать в списке `SELECT` для того, чтобы таблица была сохраняющей ключи. Например, в рассматриваемом ранее представлении объединения `V_RTLR_EMP` таблица `RETAILER` была бы таблицей, сохраняющей ключи, даже если бы столбец `RTLNR_NBR` не был включен в список оператора `SELECT`.
- И наоборот, если ключевой столбец (столбцы) таблицы представления объединения включен в список `SELECT`, это не делает данную таблицу сохраняющей ключи. В представлении `V_RTLR_EMP` тот факт, что `EMP_ID` присутствует в списке оператора `SELECT`, не означает, что таблица `EMPLOYEE` сохраняет ключи.
- Свойство таблицы сохранять ключи в представлении объединения не зависит от данных таблицы. Оно зависит от схемы данных и отношений между таблицами.

В последующих разделах будет рассказано о том, как использовать операторы `INSERT`, `UPDATE` и `DELETE` для представления объединения.

## Оператор `INSERT` и представление объединения

Давайте попытаемся вставить запись в таблицу `RETAILER`, применив оператор `INSERT` к представлению объединения `V_RTLR_EMP`:

```
INSERT INTO V_RTLR_EMP (RTLNR_NBR, NAME, SALESPERSON_ID)
VALUES (345, 'X-MART STORES', 7820);
```

```
1 row created.
```

Получилось. Давайте теперь попробуем выполнить следующий оператор `INSERT`, который вдобавок содержит значение для столбца из таблицы `EMPLOYEE`:

```
INSERT INTO V_RTLR_EMP (RTLNR_NBR, NAME, SALESPERSON_ID, SALES_REP)
VALUES (456, 'LEE PARK RECREATION CENTER', 7599, 'JAMES');
INSERT INTO V_RTLR_EMP (RTLNR_NBR, NAME, SALESPERSON_ID, SALES_REP)
```

ERROR at line 1:

ORA-01776: cannot modify more than one base table through a join view

Данный оператор пытается вставить значения в две таблицы (RETAILER и EMPLOYEE), а это запрещено. В операторе INSERT нельзя пытаться изменить столбцы таблицы, не сохраняющей ключи. Более того, нельзя применять оператор INSERT к представлению объединения, если при создании этого представления была использована инструкция WITH CHECK OPTION (даже если вы хотите вставлять значения только в таблицу, сохраняющую ключи). Например:

```
CREATE VIEW V_RTLR_EMP_WCO AS
SELECT C.RTLNR_NBR, C.NAME, C.CITY, C.SALESPERSON_ID, E.LNAME SALES_REP
FROM RETAILER C, EMPLOYEE E
WHERE C.SALESPERSON_ID = E.EMP_ID
WITH CHECK OPTION;
```

View created.

```
INSERT INTO V_RTLR_EMP_WCO (RTLNR_NBR, NAME, SALESPERSON_ID)
VALUES (345, 'X-MART STORES', 7820);
INSERT INTO V_RTLR_EMP_WCO (RTLNR_NBR, NAME, SALESPERSON_ID)
```

ERROR at line 1:

ORA-01733: virtual column not allowed here

Может быть, сообщение об ошибке «ORA-01733: virtual column not allowed here» не очень понятно, но оно показывает, что осуществлять вставку в данное представление не разрешено.

## Оператор DELETE и представление объединения

Операция DELETE может совершаться над представлением объединения, имеющим одну и только одну таблицу, сохраняющую ключи. Обсуждаемое ранее представление V\_RTLR\_EMP имеет только одну таблицу, сохраняющую ключи, — RETAILER; следовательно, для него разрешено удаление, например:

```
DELETE FROM V_RTLR_EMP
WHERE RTLNR_NBR = 214;
```

1 row deleted.

Рассмотрим другой пример с несколькими сохраняющими ключи таблицами. Создадим представление на основе самообъединения, о котором мы говорили чуть раньше в этой же главе, и попытаемся осуществить удаление:

```
CREATE VIEW V_DEPT_TEAM AS
SELECT D1.NAME TEAM1, D2.NAME TEAM2
```



```
FROM DEPARTMENT D1, DEPARTMENT D2
WHERE D1.DEPT_ID > D2.DEPT_ID;
```

View created.

```
DELETE FROM V_DEPT_TEAM
WHERE TEAM1 = 'SALES';
DELETE FROM V_DEPT_TEAM
```

ERROR at line 1:

ORA-01752: cannot delete from view without exactly one key-preserved table

## Оператор UPDATE и представление объединения

Операция UPDATE может быть применена к представлению объединения, если предполагается обновить столбец таблицы, сохраняющей ключи. Например:

```
UPDATE V_RTLR_EMP
SET NAME = 'PRO SPORTS'
WHERE RTLNR_NBR = 214;
```

1 row updated.

Данный оператор UPDATE удалось выполнить, потому что он обновляет столбец NAME таблицы RETAILER, которая сохраняет ключи. А вот приведенный ниже оператор не выполнится, так как пытается изменить столбец SALES\_REP, который относится к таблице EMPLOYEE, не сохраняющей ключи:

```
UPDATE V_RTLR_EMP
SET SALES_REP = 'ANDREW'
WHERE RTLNR_NBR = 214;
SET SALES_REP = 'ANDREW'
```

ERROR at line 2:

ORA-01779: cannot modify a column which maps to a non key-preserved table

Кроме того, возможность модифицировать представление объединения ограничивается инструкцией WITH CHECK OPTION. Если при создании представления объединения применялась инструкция WITH CHECK OPTION, вы не можете изменить ни один из столбцов объединения и ни один из столбцов таблиц:

```
UPDATE V_RTLR_EMP_WCO
SET SALESPERSON_ID = 7784
WHERE RTLNR_NBR = 214;
SET SALESPERSON_ID = 7784
```

ERROR at line 2:

ORA-01733: virtual column not allowed here

Сообщение об ошибке «ORA-01733: virtual column not allowed here» означает, что обновлять данный столбец запрещено.



## Представления словаря данных для поиска обновляемых столбцов

Oracle поддерживает представление словаря данных `USER_UPDATABLE_COLUMNS`, которое показывает все обновляемые столбцы всех таблиц и представлений схемы пользователя. Оно может быть полезно в тех случаях, когда вы хотите обновить какое-то представление, но не уверены в том, что для него разрешено обновление. `USER_UPDATABLE_COLUMNS` определяется следующим образом:

DESC USER_UPDATABLE_COLUMNS		
Name	Null?	Type
-----		
OWNER	NOT NULL	VARCHAR2(30)
TABLE_NAME	NOT NULL	VARCHAR2(30)
COLUMN_NAME	NOT NULL	VARCHAR2(30)
UPDATABLE		VARCHAR2(3)
INSERTABLE		VARCHAR2(3)
DELETABLE		VARCHAR2(3)



`ALL_UPDATABLE_COLUMNS` показывает все представления, доступ к которым вы можете получить (а не только ваши собственные), а `DBA_UPDATABLE_COLUMNS` (доступное только для администраторов базы данных) выводит все представления базы данных.

В следующем примере приводится запрос списка изменяемых столбцов представления `V_RTLR_EMP_WCO`:

```
SELECT * FROM USER_UPDATABLE_COLUMNS
WHERE TABLE_NAME = 'V_RTLR_EMP_WCO';
```

OWNER	TABLE_NAME	COLUMN_NAME	UPD	INS	DEL
-----					
DEMO	V_RTLR_EMP_WCO	RTL_R_NBR	YES	YES	YES
DEMO	V_RTLR_EMP_WCO	NAME	YES	YES	YES
DEMO	V_RTLR_EMP_WCO	CITY	YES	YES	YES
DEMO	V_RTLR_EMP_WCO	SALESPERSON_ID	NO	NO	NO
DEMO	V_RTLR_EMP_WCO	SALES_REP	NO	NO	NO

Сравните изменяемые столбцы представлений `V_RTLR_EMP_WCO` и `V_RTLR_EMP`:

```
SELECT * FROM USER_UPDATABLE_COLUMNS
WHERE TABLE_NAME = 'V_RTLR_EMP';
```

OWNER	TABLE_NAME	COLUMN_NAME	UPD	INS	DEL
-----					
DEMO	V_RTLR_EMP	RTL_R_NBR	YES	YES	YES
DEMO	V_RTLR_EMP	NAME	YES	YES	YES
DEMO	V_RTLR_EMP	CITY	YES	YES	YES
DEMO	V_RTLR_EMP	SALESPERSON_ID	YES	YES	YES
DEMO	V_RTLR_EMP	SALES_REP	NO	NO	NO

Обратите внимание на то, что в представлении V\_RTLR\_EMP столбец SALESPERSON\_ID является обновляемым, а вот в представлении V\_RTLR\_EMP\_WCO — нет.

## Синтаксис объединения стандарта ANSI в Oracle9i

Oracle9i вводит новый синтаксис объединения, соответствующий стандарту ANSI SQL, определенному для SQL/92. До версии Oracle9i Oracle поддерживал синтаксис объединения, определенный в стандарте SQL/86. Кроме того, Oracle поддерживал внешние объединения с использованием собственного оператора внешнего объединения (+), который обсуждался ранее в этой главе. Старый синтаксис объединения и принадлежащий Oracle оператор внешнего объединения продолжают поддерживаться в Oracle9i. Но стандарт ANSI вводит также несколько новых ключевых слов и новых способов задания объединений и их условий.

### Новый синтаксис объединения

При использовании традиционного синтаксиса объединения вы указываете в инструкции FROM несколько таблиц, разделяя их запятыми, например:

```
SELECT L.LOCATION_ID, D.NAME, L.REGIONAL_GROUP
FROM DEPARTMENT D, LOCATION L
WHERE D.LOCATION_ID = L.LOCATION_ID;
```

В новом синтаксисе Oracle9i вы указываете в инструкции FROM тип объединения вместе с ключевым словом JOIN. Например, чтобы выполнить внутреннее объединение таблиц DEPARTMENT и LOCATION, вы пишете:

```
FROM DEPARTMENT D INNER JOIN LOCATION L
```

В традиционном синтаксисе объединения условие объединения задается в инструкции WHERE. В новом синтаксисе Oracle9i инструкция WHERE предназначена только для фильтрации. Условие объединения отделено от инструкции WHERE и помещено в новую инструкцию ON, входящую в состав инструкции FROM. Используя новый синтаксис, перепишем условие объединения из предыдущего примера:

```
ON D.LOCATION_ID = L.LOCATION_ID;
```

Весь запрос в новом формате будет выглядеть так:

```
SELECT L.LOCATION_ID, D.NAME, L.REGIONAL_GROUP
FROM DEPARTMENT D INNER JOIN LOCATION L
ON D.LOCATION_ID = L.LOCATION_ID;
```



Задание условия объединения упрощается, если:

- вы используете объединения по равенству;
- названия столбцов двух таблиц совпадают.

Если эти два условия выполнены, для записи условия объединения можно применить новую инструкцию `USING`. В предыдущем примере рассматривалось объединение по равенству. Столбец, участвующий в условии объединения (`LOCATION_ID`), одинаково называется в обеих таблицах. Поэтому условие данного объединения можно также записать следующим образом:

```
FROM DEPARTMENT D INNER JOIN LOCATION L
  USING (LOCATION_ID);
```

Инструкция `USING` влияет и на семантику инструкции `SELECT`. Инструкция `USING` сообщает Oracle, что таблицы в объединении имеют одинаковые названия для упомянутого в инструкции `USING` столбца. Oracle объединяет эти два столбца и распознает только один такой столбец. Если вы включаете столбец объединения в список `SELECT`, Oracle не разрешает указывать для него название таблицы или псевдоним. Поэтому в нашем случае оператор `SELECT` должен выглядеть следующим образом:

```
SELECT LOCATION_ID, D.NAME, L.REGIONAL_GROUP
```

В итоге получаем:

```
SELECT LOCATION_ID, D.NAME, L.REGIONAL_GROUP
FROM DEPARTMENT D INNER JOIN LOCATION L
  USING (LOCATION_ID);
```

При попытке указать название таблицы или ее псевдоним для столбца объединения в списке `SELECT` будет выдана ошибка:

```
SELECT L.LOCATION_ID, D.NAME, L.REGIONAL_GROUP
FROM DEPARTMENT D INNER JOIN LOCATION L
  USING (LOCATION_ID);
SELECT L.LOCATION_ID, D.NAME, L.REGIONAL_GROUP
      *
ERROR at line 1:
ORA-25154: column part of USING clause cannot have qualifier
```



Поведение инструкции `USING` контрастирует с традиционным синтаксисом объединения, при использовании которого необходимо для столбцов с идентичными названиями указывать название или псевдоним таблицы.

Если условие объединения затрагивает несколько столбцов, все они указываются в инструкции `ON` и разделяются операторами `AND`. Например, если таблицы `A` и `B` объединяются на основе столбцов `c1` и `c2`, условие объединения записывается так:

```
SELECT ...
```



```
FROM A INNER JOIN B  
ON A.c1 = B.c1 AND A.c2 = B.c2
```

Если названия столбцов в обеих таблицах одинаковы, вы можете использовать инструкцию `USING` и в ней указать все столбцы, разделив их запятыми. Предыдущее объединение может быть записано следующим образом:

```
SELECT ...  
FROM A INNER JOIN B  
USING (c1, c2)
```

## Перекрестные объединения

Преимущество нового синтаксиса объединения в том, что вы не можете непредумышленно сгенерировать декартово произведение, опустив условие объединения. Но что если декартово произведение действительно необходимо? Придется вернуться к старому синтаксису? Это возможно, но лучше явно задать перекрестное объединение. Термин *перекрестное объединение* (*cross join*) — это просто еще одно название для декартова произведения.

В Oracle9i можно явно запросить перекрестное объединение, используя ключевые слова `CROSS JOIN`:

```
SELECT *  
FROM A CROSS JOIN B;
```

Польза этого нового синтаксиса в том, что он делает явным запрос перекрестного объединения. Обычно декартовы произведения являются ошибочными, и программисты, которые впоследствии будут сопровождать ваш продукт, могут столкнуться с необходимостью исправлять такие ошибки. Явное указание `CROSS JOIN` покажет им, что в данном случае декартово произведение использовано не случайно.



Новый синтаксис объединения не дает возможности забыть указать условие выполнения объединения, тем самым предотвращается случайное порождение декартовых произведений. Когда вы используете любое из новых ключевых слов объединения в инструкции `FROM`, вы сообщаете Oracle, что собираетесь выполнить объединение, и Oracle настаивает на задании условия объединения в инструкции `ON` или `USING`.

## ANSI-синтаксис внешнего объединения

Ранее в этой главе рассматривался традиционный для Oracle синтаксис внешнего объединения. Синтаксис внешнего объединения ANSI не

использует в условии объединения оператор (+). Вместо этого в инструкции FROM указывается тип объединения. Синтаксис внешнего объединения ANSI таков:

```
FROM table1 { LEFT | RIGHT | FULL } [OUTER] JOIN table2
```

Приведем перечень элементов конструкции:

*table1, table2*

Указываются таблицы, для которых выполняется внешнее объединение.

### **LEFT**

Указывает, что при получении результата необходимо использовать все строки таблицы *table1*. Для тех строк таблицы *table1*, которым не соответствуют строки *table2*, в столбцах результирующего множества, соответствующих таблице *table2*, возвращается NULL. Это эквивалентно применению (+) в условии объединения со стороны *table2* в традиционном синтаксисе.

### **RIGHT**

Указывает, что при получении результата необходимо использовать все строки таблицы *table2*. Для тех строк *table2*, которым не соответствуют строки таблицы *table1*, в столбцах результирующего множества, соответствующих таблице *table1*, возвращается NULL. Это эквивалентно применению (+) в условии объединения со стороны *table1* в традиционном синтаксисе.

### **FULL**

Указывает, что при получении результата необходимо использовать все строки таблиц *table1* и *table2*. Для тех строк *table1*, которым не соответствуют строки *table2*, в столбцах результирующего множества, соответствующих таблице *table2*, возвращается NULL. Кроме того, для тех строк таблицы *table2*, которым не соответствуют строки *table1*, в столбцах результирующего множества, соответствующих таблице *table1*, возвращается NULL. В традиционном синтаксисе не существует эквивалента FULL OUTER JOIN.

### **OUTER**

Указывает, что выполняется внешнее (outer) объединение. Это ключевое слово является необязательным. Если применены LEFT, RIGHT или FULL, Oracle автоматически считает объединение внешним. OUTER существует для полноты и дополняет ключевое слово INNER.

Чтобы выполнить левое внешнее объединение таблиц DEPARTMENT и LOCATION, можно использовать следующий оператор:

```
SELECT D.DEPT_ID, D.NAME, L.REGIONAL_GROUP  
FROM DEPARTMENT D LEFT OUTER JOIN LOCATION L  
ON D.LOCATION_ID = L.LOCATION_ID;
```

DEPT_ID	NAME	REGIONAL_GROUP
-----		
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
12	RESEARCH	NEW YORK
13	SALES	NEW YORK
14	OPERATIONS	NEW YORK
23	SALES	DALLAS
24	OPERATIONS	DALLAS
34	OPERATIONS	CHICAGO
43	SALES	BOSTON
50	MARKETING	
60	CONSULTING	

13 rows selected.

Запрос выводит все строки таблицы DEPARTMENT и соответствующие им местоположения из таблицы LOCATION. Для тех строк DEPARTMENT, которым не сопоставлены строки LOCATION, в столбце L.REGIONAL\_GROUP результирующего множества возвращается NULL. Эквивалентом является такой традиционный запрос с применением внешнего объединения:

```
SELECT D.DEPT_ID, D.NAME, L.REGIONAL_GROUP
FROM DEPARTMENT D, LOCATION L
WHERE D.LOCATION_ID = L.LOCATION_ID (+);
```

Чтобы выполнить правое внешнее объединение таблиц DEPARTMENT и LOCATION, можно использовать оператор:

```
SELECT D.DEPT_ID, D.NAME, L.REGIONAL_GROUP
FROM DEPARTMENT D RIGHT OUTER JOIN LOCATION L
ON D.LOCATION_ID = L.LOCATION_ID;
```

DEPT_ID	NAME	REGIONAL_GROUP
-----		
10	ACCOUNTING	NEW YORK
12	RESEARCH	NEW YORK
14	OPERATIONS	NEW YORK
13	SALES	NEW YORK
30	SALES	CHICAGO
34	OPERATIONS	CHICAGO
20	RESEARCH	DALLAS
23	SALES	DALLAS
24	OPERATIONS	DALLAS
		SAN FRANCISCO
40	OPERATIONS	BOSTON
43	SALES	BOSTON

12 rows selected.



Этот запрос выводит все строки таблицы LOCATION и соответствующие им подразделения из таблицы DEPARTMENT. Для тех строк LOCATION, которым не сопоставлены строки DEPARTMENT, в столбцах D.DEPT\_ID и D.NAME результирующего множества возвращается NULL. Эквивалентом является такой традиционный запрос с применением внешнего объединения:

```
SELECT D.DEPT_ID, D.NAME, L.REGIONAL_GROUP
FROM DEPARTMENT D, LOCATION L
WHERE D.LOCATION_ID (+) = L.LOCATION_ID;
```

Если вы хотите включить в результат и подразделения без местоположения, и местоположения без подразделений, необходимо полное внешнее объединение:

```
SELECT D.DEPT_ID, D.NAME, L.REGIONAL_GROUP
FROM DEPARTMENT D FULL OUTER JOIN LOCATION L
ON D.LOCATION_ID = L.LOCATION_ID;
```

DEPT_ID	NAME	REGIONAL_GROUP
10	ACCOUNTING	NEW YORK
12	RESEARCH	NEW YORK
13	SALES	NEW YORK
14	OPERATIONS	NEW YORK
20	RESEARCH	DALLAS
23	SALES	DALLAS
24	OPERATIONS	DALLAS
30	SALES	CHICAGO
34	OPERATIONS	CHICAGO
40	OPERATIONS	BOSTON
43	SALES	BOSTON
50	MARKETING	
60	CONSULTING	
		SAN FRANCISCO

14 rows selected.

Ранее в этой главе говорилось, что нельзя реализовать полное внешнее объединение, используя оператор (+) по обе стороны условия объединения. В разделе «Полные внешние объединения» показывалось, что можно обойти это ограничение, используя запрос UNION. В новом синтаксисе Oracle9i нет необходимости применять UNION для выполнения полного внешнего объединения. Новый синтаксис не только соответствует ANSI, он еще и элегантен и эффективен!

## Преимущества нового синтаксиса объединения

Разработчикам, которые привыкли к применению традиционного синтаксиса объединения Oracle, в том числе к оператору внешнего объединения (+), новый синтаксис может показаться не очень полезным. Но в его использовании есть несколько плюсов:

- Новый синтаксис объединения придерживается стандарта ANSI, что делает ваш код более переносимым.
- Новые инструкции `ON` и `USING` помогают отделить условия объединения от других условий фильтрации в инструкции `WHERE`. Это увеличивает производительность разработки и эксплуатационную надежность кода.
- Новый синтаксис делает возможным выполнение полного внешнего объединения без слияния (с помощью `UNION`) результатов двух запросов `SELECT`.

Мы рекомендуем при работе с Oracle9i использовать не традиционный, а новый синтаксис объединения.

# 4

## Групповые операции

В повседневной работе SQL-программист часто имеет дело с групповыми операциями. Используя SQL для доступа к базе данных, вы часто задаете вопросы, подобные перечисленным ниже:

- Какова максимальная заработная плата в данном подразделении?
- Сколько в каждом подразделении менеджеров?
- Сколько заказчиков существует для каждого продукта?
- Можно ли вывести среднемесячное значение продаж для каждого региона?

Для ответа на такие вопросы необходимы групповые операции. Oracle предоставляет широкий спектр возможностей по обработке групповых операций, в том числе обобщающие функции, инструкции GROUP BY и HAVING, функцию GROUPING и расширения инструкции GROUP BY: ROLLUP и CUBE.



В данной главе будут рассмотрены только простые групповые операции, использующие обобщающие функции, инструкции GROUP BY и HAVING. Более сложные групповые операции, такие как GROUPING, ROLLUP и CUBE, обсуждаются в главе 12.

## Обобщающие функции

Если говорить по существу, *обобщающая функция (aggregate function)* суммирует результаты выражения для некоторого количества строк, возвращая одно значение. Синтаксис большинства обобщающих функций таков:

```
обобщающая_функция([DISTINCT | ALL] выражение)
```



Приведем перечень элементов конструкции:

### *обобщающая функция*

Указывает имя функции, например SUM, COUNT, AVG, MAX, MIN и др.

### **DISTINCT**

Указывает, что обобщающая функция должна учитывать только неповторяющиеся значения *выражения*.

### **ALL**

Указывает, что обобщающая функция должна учитывать все значения *выражения*, в том числе и все дублирующиеся. По умолчанию считается, что использовано ALL.

### *выражение*

Указывает столбец или любое другое выражение, по которому необходимо выполнить обобщение.

Давайте рассмотрим простой пример. Для нахождения максимальной зарплаты сотрудников SQL-оператор использует функцию MAX:

```
SELECT MAX(SALARY) FROM EMPLOYEE;
```

```
MAX(SALARY)
```

```
-----  
5000
```

В последующих разделах будет рассмотрен ряд более сложных примеров, иллюстрирующих различные аспекты поведения обобщающих функций. В этих примерах будет использоваться таблица CUST\_ORDER, выглядящая следующим образом:

DESC CUST\_ORDER

Name	Null?	Type
ORDER_NBR	NOT NULL	NUMBER(7)
CUST_NBR	NOT NULL	NUMBER(5)
SALES_EMP_ID	NOT NULL	NUMBER(5)
SALE_PRICE		NUMBER(9,2)
ORDER_DT	NOT NULL	DATE
EXPECTED_SHIP_DT	NOT NULL	DATE
CANCELLED_DT		DATE
SHIP_DT		DATE
STATUS		VARCHAR2(20)

```
SELECT ORDER_NBR, CUST_NBR, SALES_EMP_ID, SALE_PRICE,  
ORDER_DT, EXPECTED_SHIP_DT  
FROM CUST_ORDER;
```

ORDER_NBR	CUST_NBR	SALES_EMP_ID	SALE_PRICE	ORDER_DT	EXPECTED_
1001	231	7354	99	22-JUL-01	23-JUL-01
1000	201	7354		19-JUL-01	24-JUL-01

1002	255	7368	12-JUL-01	25-JUL-01
1003	264	7368	56 16-JUL-01	26-JUL-01
1004	244	7368	34 18-JUL-01	27-JUL-01
1005	288	7368	99 22-JUL-01	24-JUL-01
1006	231	7354	22-JUL-01	28-JUL-01
1007	255	7368	25 20-JUL-01	22-JUL-01
1008	255	7368	25 21-JUL-01	23-JUL-01
1009	231	7354	56 18-JUL-01	22-JUL-01
1012	231	7354	99 22-JUL-01	23-JUL-01
1011	201	7354	19-JUL-01	24-JUL-01
1015	255	7368	12-JUL-01	25-JUL-01
1017	264	7368	56 16-JUL-01	26-JUL-01
1019	244	7368	34 18-JUL-01	27-JUL-01
1021	288	7368	99 22-JUL-01	24-JUL-01
1023	231	7354	22-JUL-01	28-JUL-01
1025	255	7368	25 20-JUL-01	22-JUL-01
1027	255	7368	25 21-JUL-01	23-JUL-01
1029	231	7354	56 18-JUL-01	22-JUL-01

20 rows selected.

## Обобщающие функции и NULL

Обратите внимание на то, что столбцу SALE\_PRICE таблицы CUST\_ORDER разрешено содержать значения NULL и что они присутствуют в некоторых его строках. Чтобы посмотреть, как NULL влияет на обобщающую функцию, выполним следующий оператор SQL:

```
SELECT COUNT(*), COUNT(SALE_PRICE) FROM CUST_ORDER;
```

```
COUNT(*) COUNT(SALE_PRICE)
```

20

14

Видите разницу значений, выведенных COUNT(\*) и COUNT(SALE\_PRICE)? Дело в том, что COUNT(SALE\_PRICE) игнорирует NULL-значения, в то время как COUNT(\*) этого не делает. COUNT(\*) не игнорирует NULL, потому что она считает строки, а не значения столбца. Понятие NULL не относится ко всей строке целиком. Кроме COUNT(\*) есть еще всего одна обобщающая функция, не пропускающая NULL, — GROUPING. Все остальные обобщающие функции не учитывают NULL. О функции GROUPING мы поговорим в главе 12. Сейчас же посмотрим на то, как наличие значений NULL влияет на функции, которые их игнорируют.

SUM, MAX, MIN, AVG и др. — все они пропускают NULL. Поэтому если, например, нужно вычислить такое значение, как средняя отпускная цена в таблице CUST\_ORDER, то полученная величина будет средней для 14 строк, содержащих значение в данном столбце. В приведенном ниже примере выводится общее количество строк, общая отпускная цена и средняя отпускная цена:

```
SELECT COUNT(*), SUM(SALE_PRICE), AVG(SALE_PRICE)
FROM CUST_ORDER;
```

COUNT(*)	SUM(SALE_PRICE)	AVG(SALE_PRICE)
20	788	56.2857143

Заметьте, что значение `AVG(SALE_PRICE)` не равно `SUM(SALE_PRICE)/COUNT(*)`. Ведь если бы это было так, то результат вычисления `AVG(SALE_PRICE)` равнялся бы  $788 / 20 = 39,4$ . Но так как функция `AVG` игнорирует значения `NULL`, она делит общую отпускную цену не на 20, а на 14:  $(788 / 14 = 56,2857143)$ .

Бывают случаи, когда необходимо вычислить среднее значение по всем строкам, а не только по содержащим отличные от `NULL` значения в рассматриваемом столбце. В таких ситуациях внутри вызова функции `AVG` применяется функция `NVL`, которая заменяет в столбце все значения `NULL` на 0 (или какое-либо другое приемлемое значение). (Вместо `NVL` можно также использовать `DECODE` или новую функцию `COALESCE`. Дополнительную информацию можно найти в главе 9.) Рассмотрим пример:

```
SELECT AVG(NVL(SALE_PRICE,0)) FROM CUST_ORDER;
```

AVG(NVL(SALE_PRICE,0))
39.4

Благодаря применению `NVL` при вычислении среднего значения в расчет принимаются все 20 строк, при этом строки, содержащие `NULL`-значения, учитываются как имеющие нулевое значение в столбце `SALE_PRICE`.

## Использование DISTINCT и ALL

Большинство обобщающих функций разрешают использование ключевых слов `DISTINCT` или `ALL` наряду с выражением-аргументом. `DISTINCT` дает возможность не рассматривать повторяющиеся значения, в то время как `ALL` включает дубликаты в результат. Заметьте, что в столбце `CUST_NBR` есть одинаковые значения. Посмотрим на результат выполнения следующего оператора:

```
SELECT COUNT(CUST_NBR), COUNT(DISTINCT CUST_NBR), COUNT(ALL CUST_NBR)
FROM CUST_ORDER;
```

COUNT(CUST_NBR)	COUNT(DISTINCT CUST_NBR)	COUNT(ALL CUST_NBR)
20	6	20

В столбце `CUST_NBR` имеется шесть отличных друг от друга значений. Поэтому `COUNT(DISTINCT CUST_NBR)` возвращает 6, а вот `COUNT(CUST_NBR)` и `COUNT(ALL CUST_NBR)` возвращают 20. `ALL` – значение по умолчанию, то есть если перед выражением-аргументом обобщающей функции не



указано ни DISTINCT, ни ALL, то функция будет рассматривать все строки, для которых значение выражения отлично от NULL.

Необходимо отметить, что использование ALL не означает, что обобщающая функция будет учитывать NULL. Например, COUNT(ALL SALE\_PRICE) все равно возвращает 14, а не 20:

```
SELECT COUNT(ALL SALE_PRICE) FROM CUST_ORDER;
```

```
COUNT(ALLSALE_PRICE)
```

```
-----  
14
```

Так как ALL — значение по умолчанию, то можно явно использовать его с любой обобщающей функцией. Тем не менее обобщающие функции, принимающие на входе более одного аргумента, не допускают использования DISTINCT. Это относится к CORR, COVAR\_POP, COVAR\_SAMP и всем функциям линейной регрессии.

Более того, некоторые функции, принимающие на входе один аргумент, также не разрешают использовать ключевое слово DISTINCT. К этой категории относятся STDDEV\_POP, STDDEV\_SAMP, VAR\_POP, VAR\_SAMP и GROUPING.

Если вы попытаетесь использовать ключевое слово DISTINCT в не допускающей этого функции, то получите сообщение об ошибке. Например:

```
SELECT STDDEV_POP(DISTINCT SALE_PRICE)
```

```
FROM CUST_ORDER;
```

```
SELECT STDDEV_POP(DISTINCT SALE_PRICE)
```

```
ERROR at line 1:
```

```
ORA-30482: DISTINCT option not allowed for this function
```

Использование в той же самой функции ключевого слова ALL не вызовет ошибки:

```
SELECT STDDEV_POP(ALL SALE_PRICE)
```

```
FROM CUST_ORDER;
```

```
STDDEV_POP(ALLSALE_PRICE)
```

```
-----  
29.5282639
```

## Инструкция GROUP BY

Инструкция GROUP BY, используемая совместно с обобщающими функциями, разбивает результирующее множество на несколько групп, а затем для каждой группы выдается одна строка сводной информации. Например, если нужно вычислить общее количество заказов каждого клиента, выполним следующий запрос:

```
SELECT CUST_NBR, COUNT(ORDER_NBR)  
FROM CUST_ORDER
```

GROUP BY CUST\_NBR;

CUST_NBR	COUNT(ORDER_NBR)
201	2
231	6
244	2
255	6
264	2
288	2

6 rows selected.

Запрос выдает одну сводную строку для каждого клиента. В этом заключается суть запроса GROUP BY. Мы просим Oracle сгруппировать (GROUP) результаты по номеру клиента (BY CUST\_NBR), поэтому для каждого уникального значения CUST\_NBR порождается одна строка вывода. Каждое значение для определенного клиента представляет собой сводную информацию по всем строкам данного клиента.

Необобщенное выражение CUST\_NBR из списка SELECT присутствует и в инструкции GROUP BY. Если в списке SELECT присутствует смесь обобщенных и необобщенных значений, SQL считает, что вы собираетесь выполнить операцию GROUP BY, поэтому все необобщенные выражения должны быть указаны и в инструкции GROUP BY. Если этого не сделать, SQL выдаст сообщение об ошибке. Например, если опустить инструкцию GROUP BY, возникнет следующая ошибка:

```
SELECT CUST_NBR, SALES_EMP_ID, COUNT(ORDER_NBR)
FROM CUST_ORDER;
SELECT CUST_NBR, SALES_EMP_ID, COUNT(ORDER_NBR)
```

```
*
ERROR at line 1:
ORA-00937: not a single-group group function
```

Аналогично, если не включить все необобщенные выражения списка SELECT в инструкцию GROUP BY, то SQL выдаст такую ошибку:

```
SELECT CUST_NBR, SALES_EMP_ID, COUNT(ORDER_NBR)
FROM CUST_ORDER
GROUP BY CUST_NBR;
SELECT CUST_NBR, SALES_EMP_ID, COUNT(ORDER_NBR)
```

```
*
ERROR at line 1:
ORA-00979: not a GROUP BY expression
```

Наконец, не разрешено использование групповой (обобщающей) функции в инструкции GROUP BY. При попытке такого использования, как в приведенном ниже примере, вы получите следующее сообщение об ошибке:

```
SELECT CUST_NBR, COUNT(ORDER_NBR)
FROM CUST_ORDER
```

```
GROUP BY CUST_NBR, COUNT(ORDER_NBR);
GROUP BY CUST_NBR, COUNT(ORDER_NBR)
```

ERROR at line 3:

ORA-00934: group function is not allowed here

Если в списке SELECT содержится константа, необязательно включать ее в инструкцию GROUP BY. Но включение константы в инструкцию GROUP BY не повлияет на результат. Поэтому два следующих оператора возвращают одинаковые результаты:

```
SELECT 'CUSTOMER', CUST_NBR, COUNT(ORDER_NBR)
FROM CUST_ORDER
GROUP BY CUST_NBR;
```

```
SELECT 'CUSTOMER', CUST_NBR, COUNT(ORDER_NBR)
FROM CUST_ORDER
GROUP BY 'CUSTOMER', CUST_NBR;
```

'CUSTOME	CUST_NBR	COUNT(ORDER_NBR)
CUSTOMER	201	2
CUSTOMER	231	6
CUSTOMER	244	2
CUSTOMER	255	6
CUSTOMER	284	2
CUSTOMER	288	2

8 rows selected.

Возможны случаи, когда хотелось бы, чтобы выражение присутствовало в списке для выборки, но группировка по нему не нужна. Например, если вы хотите вывести номер строки и сводную информацию для каждого клиента. Используя приведенный ниже запрос, вы получите сообщение об ошибке:

```
SELECT ROWNUM, CUST_NBR, COUNT(ORDER_NBR)
FROM CUST_ORDER
GROUP BY CUST_NBR;
SELECT ROWNUM, CUST_NBR, COUNT(ORDER_NBR)
```

ERROR at line 1:

ORA-00979: not a GROUP BY expression

Если же включить ROWNUM в инструкцию GROUP BY, получим неожиданный результат:

```
SELECT ROWNUM, CUST_NBR, COUNT(ORDER_NBR)
FROM CUST_ORDER
GROUP BY ROWNUM, CUST_NBR;
```

ROWNUM	CUST_NBR	COUNT(ORDER_NBR)
1	231	1
2	201	1



3	255	1
4	264	1
5	244	1
6	288	1
7	231	1
8	255	1
9	255	1
10	231	1
11	231	1
12	201	1
13	255	1
14	264	1
15	244	1
16	288	1
17	231	1
18	255	1
19	255	1
20	231	1

20 rows selected.

Это не то, чего мы хотели, не так ли? Требовалось получить одну итоговую строку для каждого клиента и вывести ROWNUM для таких строк. Но если ROWNUM включается в инструкцию GROUP BY, то порождается по одной итоговой строке для каждой строки, выбранной из таблицы CUST\_ORDER. Чтобы получить ожидаемый результат, необходимо использовать следующий оператор SQL:

```
SELECT ROWNUM, V.*
FROM (SELECT CUST_NBR, COUNT(ORDER_NBR)
      FROM CUST_ORDER GROUP BY CUST_NBR) V;
```

ROWNUM	CUST_NBR	COUNT(ORDER_NBR)
1	201	2
2	231	6
3	244	2
4	255	6
5	264	2
6	288	2

6 rows selected.

Конструкция внутри инструкции FROM называется встроенным представлением. О встроенных представлениях читайте в главе 5.

Включение всех выражений инструкции GROUP BY в список SELECT не является синтаксически обязательным. Но выражения, не включенные в список SELECT, не будут представлены при выводе, что может сделать вывод малоинформативным. Например:

```
SELECT COUNT(ORDER_NBR)
FROM CUST_ORDER
GROUP BY CUST_NBR;
```

```
COUNT(ORDER_NBR)
```

```
-----  
2  
6  
2  
6  
2  
2
```

```
6 rows selected.
```

Этот запрос выводит количество заказов для каждого клиента (группируя по полю CUST\_NBR), но так как значение поля CUST\_NBR в выводе не присутствует, невозможно сопоставить количество заказов конкретному клиенту. Обобщая последний пример, можно сказать, что отсутствие согласованности выражений списка SELECT и инструкции GROUP BY приводит к появлению малопонятных результатов. Следующий пример порождает вывод, который на первый взгляд кажется весьма полезным:

```
SELECT CUST_NBR, COUNT(ORDER_NBR)  
FROM CUST_ORDER  
GROUP BY CUST_NBR, ORDER_DT;
```

```
  CUST_NBR COUNT(ORDER_NBR)  
-----  
      201                2  
      231                2  
      231                4  
      244                2  
      255                2  
      255                2  
      255                2  
      264                2  
      288                2
```

```
9 rows selected.
```

Из результата видно, что была предпринята попытка подсчитать количество заказов каждого клиента. Но для некоторых значений CUST\_NBR в выводе присутствует по несколько строк. Тот факт, что в инструкцию GROUP BY был включен столбец ORDER\_DT и был сгенерирован сводный результат для каждой комбинации значений CUST\_NBR и ORDER\_DT, не нашел отражения в выводе. Смысл данных, выведенных в результате выполнения запроса, можно понять, только если посмотреть на оператор SQL. Но ведь нельзя ожидать, что все, кто будут читать выходные данные, понимают синтаксис SQL, не так ли? Поэтому рекомендуем всегда поддерживать соответствие необобщенных выражений списка SELECT выражениям инструкции GROUP BY. Более точно сформулированная версия предыдущего SQL-оператора выглядела бы так:

```
SELECT CUST_NBR, ORDER_DT, COUNT(ORDER_NBR)  
FROM CUST_ORDER  
GROUP BY CUST_NBR, ORDER_DT;
```

CUST_NBR	ORDER_DT	COUNT(ORDER_NBR)
201	19-JUL-01	2
231	18-JUL-01	2
231	22-JUL-01	4
244	18-JUL-01	2
255	12-JUL-01	2
255	20-JUL-01	2
255	21-JUL-01	2
264	16-JUL-01	2
288	22-JUL-01	2

9 rows selected.

Этот вывод согласуется с инструкцией GROUP BY запроса. Повышаются шансы на правильное трактование выходных данных.

## Инструкция GROUP BY и значения NULL

При группировке (GROUP BY) по столбцу, содержащему в некоторых строках NULL-значения, все строки с NULL-значениями помещаются в одну группу и представляются в выводе одной сводной строкой. Например,

```
SELECT SALE_PRICE, COUNT(ORDER_NBR)
FROM CUST_ORDER
GROUP BY SALE_PRICE;
```

SALE_PRICE	COUNT(ORDER_NBR)
25	4
34	2
56	4
99	4
	6

Заметьте, что в последней строке вывода в столбце SALE\_PRICE содержится NULL. Так как инструкция GROUP BY, по существу, выполняет инструкцию ORDER BY для группы по столбцам, строка со значением NULL помещается в конец. Если вы хотите, чтобы эта строка выводилась первой, можно задать ORDER BY по столбцу SALE\_PRICE в порядке убывания:

```
SELECT SALE_PRICE, COUNT(ORDER_NBR)
FROM CUST_ORDER
GROUP BY SALE_PRICE
ORDER BY SALE_PRICE DESC;
```

SALE_PRICE	COUNT(ORDER_NBR)
	6
99	4
56	4
34	2
25	4



## Инструкция GROUP BY и инструкция WHERE

При порождении итоговых результатов с использованием инструкции GROUP BY существует возможность фильтрации записей таблицы при помощи инструкции WHERE, как в приведенном ниже примере, где выводится количество заказов, в которых отпускная стоимость превышает \$25 для каждого клиента:

```
SELECT CUST_NBR, COUNT(ORDER_NBR)
FROM CUST_ORDER
WHERE SALE_PRICE > 25
GROUP BY CUST_NBR;
```

CUST_NBR	COUNT(ORDER_NBR)
231	4
244	2
264	2
288	2

При выполнении оператора SQL, содержащего инструкции WHERE и GROUP BY, Oracle сначала применяет инструкцию WHERE и отсеивает строки, не удовлетворяющие условию WHERE. Затем строки, удовлетворяющие условию WHERE, группируются в соответствии с инструкцией GROUP BY.

Синтаксис SQL требует, чтобы инструкция WHERE предшествовала инструкции GROUP BY. В противном случае будет выдана ошибка:

```
SELECT CUST_NBR, COUNT(ORDER_NBR)
FROM CUST_ORDER
GROUP BY CUST_NBR
WHERE SALE_PRICE > 25;
WHERE SALE_PRICE > 25
*
ERROR at line 4:
ORA-00933: SQL command not properly ended
```

## Инструкция HAVING

Инструкция HAVING тесно связана с инструкцией GROUP BY. Инструкция HAVING используется для наложения фильтра на группы, созданные инструкцией GROUP BY. Если запрос содержит инструкцию HAVING и инструкцию GROUP BY, результирующее множество будет содержать только те группы, которые удовлетворяют условию, указанному в инструкции HAVING. Давайте рассмотрим несколько примеров, иллюстрирующих вышесказанное. Приведенный ниже запрос возвращает количество заказов каждого клиента:

```
SELECT CUST_NBR, COUNT(ORDER_NBR)
FROM CUST_ORDER
GROUP BY CUST_NBR
```

```
HAVING CUST_NBR < 260;
```

```
CUST_NBR COUNT(ORDER_NBR)
```

CUST_NBR	COUNT(ORDER_NBR)
201	2
231	6
244	2
255	6

Заметьте, что в выводе присутствуют только клиенты с номерами, меньшими, чем 260. Это объясняется тем, что в инструкции `HAVING` указано условие `CUST_NBR < 260`. Количество заказов подсчитывается для всех клиентов, но выводятся только те группы, для которых выполнено условие инструкции `HAVING`.

Этот пример является не очень удачной иллюстрацией возможностей инструкции `HAVING`; в данном случае она просто указывает данные, которые не должны включаться в результирующее множество. Было бы эффективнее использовать `WHERE CUST_NBR < 260`, а не `HAVING CUST_NBR < 260`, так как инструкция `WHERE` исключает строки из рассмотрения до проведения группировки, а `HAVING` устраняет уже созданные группы. Правильнее было бы записать предыдущий запрос так:

```
SELECT CUST_NBR, COUNT(ORDER_NBR)
FROM CUST_ORDER
WHERE CUST_NBR < 260;
```

Следующий пример демонстрирует более удачное применение инструкции `HAVING`:

```
SELECT CUST_NBR, COUNT(ORDER_NBR)
FROM CUST_ORDER
GROUP BY CUST_NBR
HAVING COUNT(ORDER_NBR) > 2;
```

```
CUST_NBR COUNT(ORDER_NBR)
```

CUST_NBR	COUNT(ORDER_NBR)
231	6
255	6

Обратите внимание на использование в инструкции `HAVING` обобщающей функции. Здесь инструкция `HAVING` применена надлежащим образом, так как результат выполнения обобщающей функции доступен только после проведения группировки.

Синтаксис инструкции `HAVING` подобен синтаксису инструкции `WHERE`. Но для условия инструкции `HAVING` существует одно ограничение. Это условие может относиться только к выражениям списка `SELECT` или инструкции `GROUP BY`. Если указать в инструкции `HAVING` выражение, не содержащееся ни в списке `SELECT`, ни в инструкции `GROUP BY`, то в ответ будет выдано сообщение об ошибке. Например:

```
SELECT CUST_NBR, COUNT(ORDER_NBR)
FROM CUST_ORDER
```

```

GROUP BY CUST_NBR
HAVING ORDER_DT < SYSDATE;
HAVING ORDER_DT < SYSDATE
*
ERROR at line 4:
ORA-00979: not a GROUP BY expression

```

Порядок следования инструкций GROUP BY и HAVING в операторе SELECT не имеет значения. Можно определить инструкцию GROUP BY до инструкции HAVING, а можно и наоборот. Два приведенных ниже запроса совпадают друг с другом и выводят один и тот же результат:

```

SELECT CUST_NBR, COUNT(ORDER_NBR)
FROM CUST_ORDER
GROUP BY CUST_NBR
HAVING COUNT(ORDER_NBR) > 2;

```

```

SELECT CUST_NBR, COUNT(ORDER_NBR)
FROM CUST_ORDER
HAVING COUNT(ORDER_NBR) > 2
GROUP BY CUST_NBR;

```

CUST_NBR	COUNT(ORDER_NBR)
231	6
255	6

Можно использовать в одном запросе инструкции WHERE и HAVING. При этом важно понимать, как одна инструкция воздействует на другую. Имейте в виду, что WHERE выполняется первой, и строки, которые не удовлетворяют условию WHERE, не передаются в инструкцию GROUP BY. Инструкция GROUP BY сводит отфильтрованные данные в группы, и затем уже к группам применяется HAVING для устранения групп, не удовлетворяющих условию HAVING. Следующий пример иллюстрирует это:

```

SELECT CUST_NBR, COUNT(ORDER_NBR)
FROM CUST_ORDER
WHERE SALE_PRICE > 25
GROUP BY CUST_NBR
HAVING COUNT(ORDER_NBR) > 1;

```

CUST_NBR	COUNT(ORDER_NBR)
231	4
244	2
264	2
288	2

Сначала инструкция WHERE исключает из рассмотрения все заказы, для которых не выполняется условие SALE\_PRICE > 25. Остальные строки группируются по полю CUST\_NBR. Инструкция HAVING исключает из вывода клиентов, которые сделали менее двух заказов.



# 5

## Подзапросы

Иногда, прежде чем приступить собственно к делу, необходимо провести некие подготовительные операции. Например, при приготовлении пищи часто необходимо предварительно смешать несколько ингредиентов друг с другом, а затем уже соединять полученную смесь с чем-то еще. Так и некоторые типы операторов SQL выигрывают от создания в ходе выполнения промежуточного результирующего множества. Структура, ответственная за формирование промежуточного результата, называется подзапросом. В этой главе будет рассказано о подзапросах и показано, как использовать их в операторах SQL.

### Что такое подзапрос?

*Подзапрос (subquery)* – это оператор SELECT, вложенный в другой оператор SQL. Для удобства изложения будем называть содержащий подзапрос оператор SQL *охватывающим оператором (containing statement)*. Подзапросы выполняются перед выполнением охватывающего оператора SQL (исключения из этого правила описаны в разделе «Связанные подзапросы» данной главы). Результирующее множество, порожденное подзапросом, уничтожается после завершения выполнения охватывающего оператора, то есть подзапрос в рамках оператора представляет некий аналог временной таблицы.

Синтаксически подзапросы заключаются в скобки. Например, следующий оператор SELECT содержит в инструкции WHERE простой подзапрос:

```
SELECT * FROM customer  
WHERE cust_nbr = (SELECT 123 FROM dual);
```

В этом примере подзапрос до абсурда прост, и в нем нет необходимости – он использован просто для иллюстрации. При выполнении конструк-

ции первым вычисляется подзапрос. Результат этого вычисления далее рассматривается как значение для выражения инструкции WHERE:

```
SELECT * FROM customer  
WHERE cust_nbr = 123;
```

Теперь можно выполнять охватывающий запрос. В нашем случае он вернет информацию о клиенте с номером 123.

Подзапросы чаще всего встречаются в инструкции WHERE операторов SELECT, UPDATE или DELETE. Подзапрос также может быть *связанным (cor-related)* с охватывающим оператором SQL, в том смысле, что он ссылается на один или несколько его столбцов. Если же в подзапросе отсутствуют ссылки за его пределы, то он называется *несвязанным (non-cor-related)*. Менее распространенной, но столь же полезной разновидностью подзапросов является *встроенное представление (inline view)*, используемое в инструкции FROM. Встроенные представления всегда не связаны, они выполняются первыми и могут рассматриваться как кэшированные в памяти неиндексированные таблицы, используемые в основном запросе.

Подзапросы полезны тем, что позволяют выполнить сравнение, не изменяя размера результирующего множества. Например, нам требуется найти всех клиентов, делавших заказы в прошлом месяце, но мы хотим, чтобы каждый клиент упоминался только один раз, независимо от количества сделанных им заказов. И если объединение таблиц клиентов и заказов включает в результирующее множество все заказы клиента, то подзапрос по таблице заказов с помощью оператора IN или EXISTS позволит определить, размещал ли клиент заказ, не учитывая при этом количество сделанных заказов.

## Несвязанные подзапросы

Несвязанные подзапросы позволяют сравнить каждую строку охватывающего оператора SQL с набором значений. Разобьем несвязанные подзапросы на следующие три категории в зависимости от количества строк и столбцов, возвращаемых ими в результирующем множестве:

- Однострочные одностолбцовые подзапросы.
- Многострочные одностолбцовые подзапросы.
- Многостолбцовые подзапросы.

Для каждой из категорий в охватывающем SQL-операторе может использоваться свой набор операторов для взаимодействия с подзапросом.

## Однострочные одностолбцовые подзапросы

Подзапрос, который возвращает одну строку и один столбец, интерпретируется охватывающим оператором как скаляр, поэтому неудиви-



тельно, что эта категория подзапросов называется *скалярными подзапросами (scalar subqueries)*. Подзапрос может присутствовать в любой части условия, к нему применяются обычные операторы сравнения, такие как =, <, >, !=, <=, >=. Используем однострочный одностолбцовый подзапрос нахождения всех служащих, чья заработная плата превышает среднее значение. Подзапрос возвращает среднюю зарплату, а охватывающий оператор возвращает данные всех служащих, получающих больше возвращаемого подзапросом значения:

```
SELECT lname
FROM employee
WHERE salary > (SELECT AVG(salary)
                FROM EMPLOYEE);
```

```

LNAME
-----
Brown
Smith
Blake
Isaacs
Jacobs
King
Fox
Anderson
Nichols
Iverson
Peters
Russell
```

Как видите, вполне разумно, что подзапрос обращается к той же таблице, что и охватывающий запрос. В действительности подзапросы часто используются для выделения какого-то подмножества записей таблицы. Например, график технического обслуживания многих приложений содержит такую операцию, как очистка таблиц с рабочими данными, например журнала исключений или журнала загрузки. Каждую неделю специальная процедура может уничтожать все, за исключением информации за последний день, например:

```
DELETE FROM load_log
WHERE load_dt < (SELECT MAX(TRUNC(load_dt))
                FROM load_log);
```

Несвязанные подзапросы часто встречаются и вне инструкции WHERE, как в следующем запросе, определяющем, какой торговый представитель занимается наибольшим количеством заказов:

```
SELECT sales_emp_id, COUNT(*)
FROM cust_order
GROUP BY sales_emp_id
HAVING COUNT(*) = (SELECT MAX(COUNT(*))
                  FROM cust_order
                  GROUP BY sales_emp_id);
```



SALES_EMP_ID	COUNT(*)
30	121

Подзапрос вычисляет количество заказов, относящихся к каждому торговому представителю, а затем применяет функцию MAX, чтобы вернуть только максимальное количество заказов. Охватывающий оператор осуществляет такую же группировку, что и подзапрос, а затем сохраняет только тех представителей, для которых количество обслуженных заказов совпадает с максимальным значением, возвращенным подзапросом. Интересно, что охватывающий запрос может вернуть несколько строк, если несколько торговых представителей имеют одинаково большое количество заказов, в то время как подзапрос гарантированно возвращает одну строку и один столбец. Если вам кажется, что неразумно использовать одну и ту же группировку в подзапросе и в охватывающем операторе, вы абсолютно правы. В главе 13 вы узнаете о более эффективных способах обработки такого типа запросов.

Пока что вы встречались со скалярными подзапросами в инструкциях WHERE и HAVING оператора SELECT, а также в инструкции WHERE оператора DELETE. Прежде чем обратиться к другим типам подзапросов, давайте уточним, где в операторах SQL могут использоваться подзапросы, а где нет:

- Инструкция FROM может содержать любые типы несвязанных подзапросов.
- Инструкции SELECT и ORDER BY могут содержать скалярные подзапросы.
- В инструкции GROUP BY не разрешено использовать подзапросы.
- Инструкции START WITH и CONNECT BY, используемые для запроса иерархических данных, могут содержать подзапросы; об этом будет подробно рассказано в главе 8.

## Многострочные подзапросы

Вы научились применять однострочные одностолбцовые подзапросы. Теперь пришло время перейти к подзапросам, возвращающим несколько строк. Если подзапрос возвращает несколько строк, нельзя применять операторы сравнения, ведь отдельное значение невозможно напрямую сравнить с множеством. Однако *можно* сравнить отдельное значение с каждым из значений множества. Для этого с операторами сравнения применяют специальные ключевые слова ANY и ALL для определения того, равно ли значение (или меньше, больше и т. д.) какому-нибудь из членов множества (any) или всем (all) членам множества. Рассмотрим такой запрос:

```
SELECT fname, lname
FROM employee
WHERE dept_id = 3 AND salary >= ALL
(SELECT salary
```

```
FROM employee
WHERE dept_id = 3);
```

FNAME	LNAME
Mark	Russell

Подзапрос возвращает множество с данными о зарплате служащих подразделения 3, а охватывающий запрос проверяет каждого служащего этого подразделения на предмет того, является ли величина его заработной платы большей или равной по отношению к каждой зарплате, возвращенной подзапросом. В результате выводится фамилия самого высокооплачиваемого сотрудника подразделения 3. Любой сотрудник, кроме самого низкооплачиваемого, имеет зарплату, которая *>= какой-то* из зарплат подразделения, и только самый высокооплачиваемый сотрудник имеет зарплату, которая *>= всем* зарплатам подразделения. Если несколько сотрудников имеют одинаковую наибольшую зарплату, будет выведено несколько фамилий.

Можно переформулировать предыдущий запрос и искать сотрудника, зарплата которого не меньше, чем зарплата любого другого сотрудника. В этом случае применяется ключевое слово ANY:

```
SELECT fname, lname
FROM employee
WHERE dept_id = 3 AND NOT salary < ANY
(SELECT salary
 FROM employee
 WHERE dept_id = 3);
```

Почти всегда один запрос можно записать несколькими способами. При этом стоит обратить внимание на достижение разумного компромисса между производительностью и читабельностью. В данном случае можно было бы предпочесть использовать `AND salary >= ALL`, а не `AND NOT salary < ANY`, так как первый вариант проще для понимания, однако второй может оказаться более эффективным, так как каждое вычисление результатов подзапроса требует от 1 до N сравнений при использовании варианта с ANY и ровно N сравнений при использовании ALL.<sup>1</sup>

В следующем запросе оператор ANY применяется для поиска всех сотрудников, зарплата которых превышает зарплату любого руководителя высшего уровня:

```
SELECT fname, lname
FROM employee
```

---

<sup>1</sup> Если в подразделении работает 100 человек, каждая из 100 зарплат должна быть сравнена со всем множеством из 100 значений. При использовании ANY сравнение можно прекратить, как только в множестве найдена более высокая зарплата, в то время как при использовании ALL необходимы 100 сравнений, чтобы убедиться в том, что в множестве нет более низких зарплат.



```
WHERE manager_emp_id IS NOT NULL
AND salary > ANY
(SELECT salary
FROM employee
WHERE manager_emp_id IS NULL);
```

FNAME	LNAME
Laura	Peters
Mark	Russell

Подзапрос возвращает множество зарплат всех руководителей высшего уровня, а охватывающий запрос возвращает фамилии сотрудников, не относящихся к высшему уровню, зарплата которых при этом превышает любую из величин, возвращенных подзапросом. Каждый раз, когда такой запрос будет возвращать одну или более строк, можете быть уверены, что руководство примет решение об увеличении своей оплаты.

Если бы в каком-то из трех предыдущих запросов не был применен оператор ANY или ALL, была бы сгенерирована ошибка:

ORA-01427: single-row subquery returns more than one row

Описание ошибки («Однострочный запрос возвращает более чем одну строку») выглядит несколько странно. Как может однострочный запрос вернуть несколько строк? Сообщение об ошибке пытается выразить мысль о том, что был определен многострочный запрос, в то время как разрешен только однострочный. Если вы не уверены в том, что подзапрос вернет ровно одну строку, необходимо использовать ключевое слово ANY или ALL, чтобы обеспечить работоспособность кода.

Для работы с многострочными запросами можно применять не только ключевые слова ANY и ALL, но и оператор IN. Использование для подзапроса оператора IN функционально эквивалентно применению = ANY. Если в множестве, возвращенном подзапросом, найдено совпадение, IN возвращает TRUE. Используем IN для того, чтобы отложить отправку всех заказов, содержащих детали, в настоящий момент отсутствующие на складе:

```
UPDATE cust_order
SET expected_ship_dt = TRUNC(SYSDATE) + 1
WHERE ship_dt IS NULL AND order_nbr IN
(SELECT l.order_nbr
FROM line_item l, part p
WHERE l.part_nbr = p.part_nbr AND p.inventory_qty = 0);
```

Подзапрос возвращает множество заказов, в которых запрошены отсутствующие на складе детали, а охватывающий оператор UPDATE изменяет предполагаемую дату отгрузки для всех входящих в множество заказов. Думаем, вы согласны с тем, что оператор IN интуитивно понятнее, чем = ANY, поэтому в подобных ситуациях почти всегда исполь-



зуется IN. Аналогично можно использовать NOT IN вместо != ANY, как в следующем запросе, который удаляет всех клиентов, которые не разместили ни одного заказа в течение пяти последних лет:

```
DELETE FROM customer
WHERE cust_nbr NOT IN
(SELECT cust_nbr
 FROM cust_order
 WHERE order_dt >= TRUNC(SYSDATE) - (365 * 5));
```

Подзапрос возвращает множество клиентов, которые *делали* заказы в течение последних пяти лет, а охватывающий оператор DELETE удаляет всех клиентов, которые не вошли в множество, возвращенное подзапросом.

Поиск членов одного множества, которые *не* входят в другое множество, называется *антиобъединением* (*anti-join*). Из названия понятно, что это противоположность объединения – строки таблицы A возвращаются, если указанные данные *не* найдены в таблице B. Оптимизатор Oracle может применять к выполнению таких запросов различные подходы, в том числе антиобъединение *хешированием* (*hash anti-join*) и *слиянием* (*merge anti-join*).<sup>1</sup>

## Многостолбцовые подзапросы

Все ранее рассмотренные примеры сравнивали один столбец охватывающего оператора SQL с результирующим множеством подзапроса, но подзапрос может возвращать и несколько столбцов. Рассмотрим оператор UPDATE, который накапливает данные из оперативной таблицы в сводную:

```
UPDATE monthly_orders SET
tot_orders = (SELECT COUNT(*)
 FROM cust_order
 WHERE order_dt >= TO_DATE('01-NOV-2001', 'DD-MON-YYYY')
 AND order_dt < TO_DATE('01-DEC-2001', 'DD-MON-YYYY')
 AND cancelled_dt IS NULL),
max_order_amt = (SELECT MAX(sale_price)
 FROM cust_order
 WHERE order_dt >= TO_DATE('01-NOV-2001', 'DD-MON-YYYY')
 AND order_dt < TO_DATE('01-DEC-2001', 'DD-MON-YYYY')
 AND cancelled_dt IS NULL),
min_order_amt = (SELECT MIN(sale_price)
 FROM cust_order
 WHERE order_dt >= TO_DATE('01-NOV-2001', 'DD-MON-YYYY'))
```

---

<sup>1</sup> Так как это не книга по настройке СУБД, мы воздержимся от исследования внутренних механизмов работы оптимизатора Oracle и воздействия на него через хиты. Дополнительную информацию вы найдете в книге «Oracle SQL Tuning Pocket Reference» Марка Гари (Mark Gurry) издательства O'Reilly.

```

AND order_dt < TO_DATE('01-DEC-2001', 'DD-MON-YYYY')
AND cancelled_dt IS NULL),
tot_amt = (SELECT SUM(sale_price)
FROM cust_order
WHERE order_dt >= TO_DATE('01-NOV-2001', 'DD-MON-YYYY')
AND order_dt < TO_DATE('01-DEC-2001', 'DD-MON-YYYY')
AND cancelled_dt IS NULL)
WHERE month = 11 and year = 2001;

```

Оператор UPDATE изменяет четыре столбца таблицы monthly\_orders, и значения для каждого из этих четырех столбцов вычисляются путем обобщения данных из таблицы cust\_order. Присмотревшись внимательно, вы заметите, что инструкции WHERE всех четырех подзапросов одинаковы, единственным их отличием является тип обобщения данных. Следующий запрос показывает, как можно заполнить все четыре столбца за один проход таблицы cust\_order:

```

UPDATE monthly_orders
SET (tot_orders, max_order_amt, min_order_amt, tot_amt) =
(SELECT COUNT(*), MAX(sale_price), MIN(sale_price), SUM(sale_price)
FROM cust_order
WHERE order_dt >= TO_DATE('01-NOV-2001', 'DD-MON-YYYY')
AND order_dt < TO_DATE('01-DEC-2001', 'DD-MON-YYYY')
AND cancelled_dt IS NULL)
WHERE month = 11 and year = 2001;

```

Второй оператор достигает того же результата более эффективно, чем первый, выполняя четыре обобщения за один проход таблицы cust\_order вместо одного обобщения за каждый из четырех проходов.

Вы видели пример использования многостолбцового подзапроса в инструкции SET оператора UPDATE, но такие подзапросы могут использоваться и в инструкции WHERE операторов SELECT, UPDATE или DELETE. В следующем примере из открытых заказов удаляются все пункты, содержащие детали, снятые с производства:

```

DELETE FROM line_item
WHERE (order_nbr, part_nbr) IN
(SELECT c.order_nbr, p.part_nbr
FROM cust_order c, line_item li, part p
WHERE c.ship_dt IS NULL AND c.cancelled_dt IS NULL
AND c.order_nbr = li.order_nbr
AND li.part_nbr = p.part_nbr
AND p.status = 'DISCONTINUED');

```

Обратите внимание на оператор IN в инструкции WHERE. Два столбца указаны вместе в скобках перед ключевым словом IN. Значения этих двух столбцов сравниваются с множеством пар значений, возвращаемых в каждой строке результирующего множества подзапроса. Если найдено совпадение, строка удаляется из таблицы line\_item.



## Связанные подзапросы

Подзапрос, который ссылается на один или более столбцов охватывающего его оператора SQL, называется *связанным подзапросом (correlated subquery)*. В отличие от несвязанных подзапросов, которые выполняются один раз перед выполнением охватывающего оператора, связанный подзапрос выполняется по одному разу для каждой строки промежуточного результирующего множества охватывающего оператора. Рассмотрим, например, запрос, который находит все детали, поставленные Acme Industries, которые в течение декабря были проданы десять или более раз:

```
SELECT p.part_nbr, p.name
FROM supplier s, part p
WHERE s.name = 'Acme Industries'
AND s.supplier_id = p.supplier_id
AND 10 <=
  (SELECT COUNT(*)
   FROM cust_order co, line_item li
   WHERE li.part_nbr = p.part_nbr
        AND li.order_nbr = co.order_nbr
        AND co.order_dt >= TO_DATE('01-DEC-2001', 'DD-MON-YYYY'));
```

Упоминание `p.part_nbr` — это то, что делает подзапрос связанным, так как для выполнения этого подзапроса значения `p.part_nbr` должны быть предоставлены охватывающим оператором. Если в таблице деталей 10 000 записей, но только 100 из них поставяет Acme Industries, подзапрос будет выполнен один раз для каждой из 100 строк промежуточного результирующего множества, созданного при объединении таблиц деталей и поставщиков.<sup>1</sup>

Связанные подзапросы часто используются для проверки существования отношения без учета количества элементов этого отношения. Можно, например, найти все детали, которые в 2002 году были отгружены хотя бы один раз. Для таких запросов применяется оператор EXISTS:

```
SELECT p.part_nbr, p.name, p.unit_cost
FROM part p
WHERE EXISTS
  (SELECT 1 FROM line_item li, cust_order co
   WHERE li.part_nbr = p.part_nbr
        AND li.order_nbr = co.order_nbr
        AND co.ship_dt >= TO_DATE('01-JAN-2002', 'DD-MON-YYYY'));
```

Пока подзапрос возвращает одну или более строк, условие EXISTS выполняется вне зависимости от того, сколько именно строк возвращено

---

<sup>1</sup> Используя хит `PUSH_SUBQ`, можно добиться более раннего вычисления подзапроса; снова предлагаем вам обратиться за дополнительной информацией (если она вас интересует) к хорошей книге по настройке Oracle.



подзапросом. Так как оператор EXISTS возвращает TRUE или FALSE (в зависимости от того, возвращает подзапрос строки или нет), сами столбцы, возвращенные подзапросом, к делу не относятся. Однако так как для оператора SELECT требуется как минимум один столбец, общепринятой практикой в таких случаях является использование литерала «1» или специального символа «\*».

Можно реализовать проверку и на отсутствие отношения:

```
UPDATE customer c
SET c.inactive_ind = 'Y', c.inactive_dt = TRUNC(SYSDATE)
WHERE c.inactive_dt IS NULL
AND NOT EXISTS (SELECT 1 FROM cust_order co
WHERE co.cust_nbr = c.cust_nbr
AND co.order_dt > TRUNC(SYSDATE) - 365);
```

Этот оператор делает неактивными все записи о клиентах, которые за прошедший год не сделали ни одного заказа. Подобные запросы часто встречаются в программах технического обслуживания. Например, внешние ключи могут предотвратить ссылку дочерних записей на несуществующего предка, но родительские записи без дочерних вполне возможны. Если бизнес-правила не допускают подобной ситуации, можно раз в неделю запускать утилиту, которая будет удалять такие записи:

```
DELETE FROM cust_order co
WHERE co.order_dt > TRUNC(SYSDATE) - 7
AND co.cancelled_dt IS NULL
AND NOT EXISTS
(SELECT 1 FROM line_item li
WHERE li.order_nbr = co.order_nbr);
```

Запрос, в который связанный подзапрос включен при помощи оператора EXISTS, называется *частичным объединением (semi-join)*. Частичное объединение включает строки таблицы А, для которых соответствующие данные один или более раз встречаются в таблице В. То есть размер итогового результирующего множества не зависит от количества совпадений, найденных в таблице В. Как и для антиобъединений, оптимизатор Oracle может применять различные приемы для выполнения таких запросов, в том числе частичное объединение *слиянием (merge semi-join)* и *хешированием (hash semi-join)*.

Хотя они очень часто используются вместе, связанные подзапросы не требуют применения оператора EXISTS. Например, если в базе данных содержатся денормализованные столбцы, можно каждую ночь запускать программу перерасчета денормализованных данных:

```
UPDATE customer c
SET (c.tot_orders, c.last_order_dt) =
(SELECT COUNT(*), MAX(co.order_dt)
FROM cust_order co
WHERE co.cust_nbr = c.cust_nbr
AND co.cancelled_dt IS NULL);
```

Так как инструкция SET присваивает значения столбцам таблицы, единственным разрешенным оператором является «=». Подзапрос возвращает ровно одну строку (благодаря обобщающим функциям), поэтому результаты можно спокойно присваивать соответствующим столбцам. Вместо того чтобы каждый день пересчитывать всю сумму, более эффективно обновлять данные только тех клиентов, которые делали заказы в этот день:

```
UPDATE customer c SET (c.tot_orders, c.last_order_dt) =
  (SELECT c.tot_orders + COUNT(*), MAX(co.order_dt)
   FROM cust_order co
   WHERE co.cust_nbr = c.cust_nbr
        AND co.cancelled_dt IS NULL
        AND co.order_dt >= TRUNC(SYSDATE))
WHERE c.cust_nbr IN
  (SELECT co.cust_nbr FROM cust_order co
   WHERE co.order_dt >= TRUNC(SYSDATE)
        AND co.cancelled_dt IS NULL);
```

Как видно из предыдущего примера, данные охватывающего оператора могут быть использованы в связанном подзапросе не только в условии объединения инструкции WHERE. В данном примере инструкция SELECT связанного подзапроса для вычисления нового значения прибавляет сегодняшнюю сумму продаж к предыдущему значению tot\_orders таблицы customer.

## Встроенные представления

В большинстве публикаций, посвященных SQL, инструкцию FROM оператора SELECT описывают как содержащую список таблиц и/или представлений. Забудьте о таком определении, заменив его на следующее: инструкция FROM содержит список наборов данных. Под таким углом зрения легче понять, как инструкция FROM может содержать таблицы (постоянные наборы данных), представления (виртуальные наборы данных) и операторы SELECT (временные наборы данных). Оператор SELECT в инструкции FROM охватывающего оператора SELECT называется *встроенным представлением (inline view)*<sup>1</sup>: – это одно из наиболее мощных и недостаточно используемых средств Oracle SQL. Рассмотрим простой пример:

```
SELECT d.dept_id, d.name, emp_cnt.tot
FROM department d,
  (SELECT dept_id, COUNT(*) tot
   FROM employee
```

---

<sup>1</sup> По мнению авторов, термин «встроенное представление» приводит к путанице и пугает людей. Так как речь идет о подзапросе, выполняемом прежде чем охватывающий запрос, более удачным было бы название «предварительный запрос».



```
GROUP BY dept_id) emp_cnt
WHERE d.dept_id = emp_cnt.dept_id;
```

DEPT_ID	NAME	TOT
1	Human Resources	1
2	Accounting	1
3	Domestic Sales	19
4	International Sales	5

В этом примере в инструкции FROM упоминается таблица подразделений и встроенное представление emp\_cnt, которое вычисляет количество сотрудников в каждом подразделении. Два набора данных объединяются по полю dept\_id, и для каждого подразделения возвращаются идентификатор, название и количество служащих. Рассмотренный пример чрезвычайно прост, но в действительности встроенные представления позволяют делать в одном запросе такие вещи, которые без их использования потребовали бы многих операторов SELECT или применения процедурного языка.

## Базовые сведения о встроенных представлениях

Так как другие элементы охватывающего оператора ссылаются на результирующее множество встроенного представления, необходимо дать ему имя и обеспечить псевдонимами все столбцы с неоднозначными названиями. В предыдущем примере встроенное представление называлось emp\_cnt, а столбцу COUNT(\*) был присвоен псевдоним tot. Как и другие типы подзапросов, встроенное представление может объединять несколько таблиц, вызывать встроенные и пользовательские функции, задавать хинты (hints) для оптимизатора и содержать инструкции GROUP BY, HAVING и CONNECT BY. В отличие от других типов подзапросов, встроенное представление может содержать инструкцию ORDER BY, что открывает новые интересные возможности (в разделе «Разбор примера подзапроса: N лучших работников» этой главы будет приведен пример использования в подзапросе инструкции ORDER BY).

Встроенные представления особенно полезны, когда необходимо объединить данные на разных уровнях обобщения. В рассмотренном ранее примере необходимо было извлечь все строки из таблицы подразделений и включить обобщенные данные из таблицы служащих, поэтому было решено осуществить суммирование в рамках встроенного представления и объединить результаты с таблицей служащих. Любой кто занимался составлением отчетов или писал ETL-приложения для хранилищ данных (extraction, transformation, load – технология извлечения, преобразования и загрузки данных), несомненно, сталкивался с ситуациями, когда необходимо объединить данные различных уровней обобщения. Благодаря встроенным представлениям можно получить нужные результаты с помощью одного оператора SQL, вместо того чтобы разбивать логику на многочисленные фрагменты или же писать код на процедурном языке.



При работе со встроенными представлениями задайте себе следующие вопросы:

1. Как встроенное представление повлияет на читабельность и, что важнее, на производительность охватывающего оператора?
2. Насколько большим будет порожденное встроенным представлением результирующее множество?
3. Как часто мне нужно будет это конкретное множество данных?

Обычно использование встроенного представления должно улучшать читабельность и производительность запроса, и оно должно порождать легко управляемый набор данных, который не представляет ценности для других операторов и сеансов. В противном случае стоит подумать о создании постоянной или временной таблицы, чтобы данные были доступны нескольким сеансам и чтобы при необходимости можно было построить дополнительные индексы.

## Выполнение запроса

Встроенные представления всегда выполняются перед охватывающим запросом, поэтому не могут ссылаться на столбцы других таблиц или встроенных представлений того же запроса. После выполнения охватывающий запрос обращается к встроенному представлению как к неиндексированной таблице, находящейся в памяти. Если заданы вложенные встроенные представления, первым выполняется самое внутреннее представление, затем представление предыдущего уровня и т. д. Рассмотрим такой запрос:

```
SELECT d.dept_id dept_id, d.name dept_name,
       dept_orders.tot_orders tot_orders
FROM department d,
     (SELECT e.dept_id dept_id, SUM(emp_orders.tot_orders) tot_orders
      FROM employee e,
           (SELECT sales_emp_id, COUNT(*) tot_orders
            FROM cust_order
            WHERE order_dt >= TRUNC(SYSDATE) - 365
            AND cancelled_dt IS NULL
            GROUP BY sales_emp_id
           ) emp_orders
      WHERE e.emp_id = emp_orders.sales_emp_id
      GROUP BY e.dept_id
     ) dept_orders
WHERE d.dept_id = dept_orders.dept_id;
```

DEPT_ID	DEPT_NAME	TOT_ORDERS
3	Domestic Sales	666
4	International Sales	175

Если вы впервые встретились со встроенными представлениями, этот запрос мог вас напугать. Начните с самого внутреннего запроса, пой-

мите, какое результирующее множество он возвращает, затем переходите к следующему уровню. Так как встроенные представления должны быть несвязанными, можно выполнить оператор SELECT каждого встроенного представления по отдельности и посмотреть на результат.<sup>1</sup> В предыдущем примере выполнение встроенного представления emp\_orders сгенерировало бы такое результирующее множество:

```
SELECT sales_emp_id, COUNT(*) tot_orders
FROM cust_order
WHERE order_dt >= TRUNC(SYSDATE) - 365
AND cancelled_dt IS NULL
GROUP BY sales_emp_id
```

```
SALES_EMP_ID TOT_ORDERS
```

11	35
12	35
13	35
14	35
15	35
16	35
17	35
18	35
19	35
20	35
21	35
22	35
23	35
24	35
25	35
26	35
27	35
28	35
29	35
30	36
31	35
32	35
33	35
34	35

Множество emp\_orders содержит всех торговых представителей, которые занимались заказами в прошлом году, а также общее количество зарегистрированных заказов. Уровнем выше находится встроенное представление dept\_orders, которое объединяет набор данных emp\_orders с таблицей служащих и обобщает количество заказов до уровня подразделения. Результирующее множество выглядит так:

```
SELECT e.dept_id dept_id, SUM(emp_orders.tot_orders) tot_orders
FROM employee e,
     (SELECT sales_emp_id, COUNT(*) tot_orders
      FROM cust_order
```

<sup>1</sup> С точки зрения встроенного представления это «событие вне запроса».

```

WHERE order_dt >= TRUNC(SYSDATE) - 365
AND cancelled_dt IS NULL
GROUP BY sales_emp_id
) emp_orders
WHERE e.emp_id = emp_orders.sales_emp_id
GROUP BY e.dept_id

DEPT_ID TOT_ORDERS
-----
3      666
4      175

```

Наконец, множество dept\_orders объединяется с таблицей подразделений и формируется следующее множество:

```

DEPT_ID DEPT_NAME      TOT_ORDERS
-----
3 Domestic Sales      666
4 International Sales  175

```

После завершения выполнения запроса результирующие множества emp\_orders и dept\_orders отбрасываются.

## Создание наборов данных

Наряду с обращением к существующим таблицам встроенные представления могут заниматься созданием специализированных наборов данных, не существовавших в базе данных. Например, может потребоваться суммировать за предыдущий год отдельно мелкие, средние и крупные заказы, но при этом понятие величины заказа в базе данных не было определено. Можно создать таблицу с тремя записями для определения различных размеров и их границ, но эта информация нужна только для одного запроса, и не хотелось бы загромождать базу данных десятками маленьких узкоспециальных таблиц. Одним из вариантов решения является использование операторов для работы с множествами, таких как UNION,<sup>1</sup> для создания специального набора данных:

```

SELECT 'SMALL' name, 0 lower_bound, 999 upper_bound from dual
UNION ALL
SELECT 'MEDIUM' name, 1000 lower_bound, 24999 upper_bound from dual
UNION ALL
SELECT 'LARGE' name, 25000 lower_bound, 9999999 upper_bound from dual;

NAME      LOWER_BOUND UPPER_BOUND
-----
SMALL          0         999
MEDIUM      1000      24999
LARGE      25000     9999999

```

---

<sup>1</sup> Об операциях над множествами будет подробно рассказано в главе 7. Оператор UNION применяется для объединения отдельных наборов данных в единое множество.



Теперь можно заключить этот запрос во встроенное представление и использовать его для выполнения необходимого суммирования:

```
SELECT sizes.name order_size, SUM(co.sale_price) tot_dollars
FROM cust_order co,
( SELECT 'SMALL' name, 0 lower_bound, 999 upper_bound from dual
  UNION ALL
  SELECT 'MEDIUM' name, 1000 lower_bound, 24999 upper_bound from dual
  UNION ALL
  SELECT 'LARGE' name, 25000 lower_bound, 9999999 upper_bound from dual
) sizes
WHERE co.cancelled_dt IS NULL
      AND co.order_dt >= TO_DATE('01-JAN-2001', 'DD-MON-YYYY')
      AND co.order_dt < TO_DATE('01-JAN-2002', 'DD-MON-YYYY')
      AND co.sale_price BETWEEN sizes.lower_bound AND sizes.upper_bound
GROUP BY sizes.name;
```

ORDER_	TOT_DOLLARS
LARGE	7136214
MEDIUM	32395018
SMALL	63676

Будьте внимательны при создании набора диапазонов, убедитесь, что вы не оставили промежутков, через которые данные могут «ускользнуть». Например, заказ на общую сумму \$999,50 не войдет ни в мелкую, ни в среднюю категорию, так как \$999,50 не входит ни в диапазон от \$0 до \$999, ни в диапазон от \$1,000 до \$24,999. Можно было бы сделать границы диапазонов перекрывающимися, тогда все данные были бы учтены. Но в этом случае нельзя использовать BETWEEN.

```
SELECT sizes.name order_size, SUM(co.sale_price) tot_dollars
FROM cust_order co,
( SELECT 'SMALL' name, 0 lower_bound, 1000 upper_bound from dual
  UNION ALL
  SELECT 'MEDIUM' name, 1000 lower_bound, 25000 upper_bound from dual
  UNION ALL
  SELECT 'LARGE' name, 25000 lower_bound, 9999999 upper_bound from dual
) sizes
WHERE co.cancelled_dt IS NULL
      AND co.order_dt >= TO_DATE('01-JAN-2001', 'DD-MON-YYYY')
      AND co.order_dt < TO_DATE('01-JAN-2002', 'DD-MON-YYYY')
      AND co.sale_price >= sizes.lower_bound
      AND co.sale_price < sizes.upper_bound
GROUP BY sizes.name;
```

Заметьте, что между нашими сегментами нет ни пробелов, ни перекрытий, и можно быть уверенными в том, что данные не будут утеряны при суммировании.

Создание множеств данных также полезно для определения того, какие данные *не* хранятся в базе. Например, ваш руководитель может

попросить подготовить отчет с суммами продаж за каждый день 2000 года, включая дни, когда ничего не было продано. Таблица `cust_order` содержит данные за каждый день, в который производились заказы, но таблицы, содержащей записи за все дни года, в базе данных не существует. Чтобы предоставить отчет, необходимо создать базовую таблицу, содержащую запись для каждого дня 2000 года, а затем выполнить ее внешнее объединение с множеством обобщенных продаж за тот же период.

Так как год состоит из 365 или 366 дней, создадим множество  $\{0, 1, 2, \dots, 399\}$ , прибавим каждый член множества к начальной дате 1 января 2000 года и предоставим Oracle удаление строк, которые не принадлежат 2000 году. Чтобы создать множество  $\{0, 1, 2, \dots, 399\}$ , создадим наборы  $\{0, 1, 2, \dots, 10\}$ ,  $\{0, 10, 20, 30, \dots, 90\}$  и  $\{0, 100, 200, 300\}$  и сформируем декартово произведение этих множеств:

```
SELECT ones.x + tens.x + hundreds.x tot
FROM
  (SELECT 0 x FROM dual UNION ALL
   SELECT 1 x FROM dual UNION ALL
   SELECT 2 x FROM dual UNION ALL
   SELECT 3 x FROM dual UNION ALL
   SELECT 4 x FROM dual UNION ALL
   SELECT 5 x FROM dual UNION ALL
   SELECT 6 x FROM dual UNION ALL
   SELECT 7 x FROM dual UNION ALL
   SELECT 8 x FROM dual UNION ALL
   SELECT 9 x FROM dual) ones,
  (SELECT 0 x FROM dual UNION ALL
   SELECT 10 x FROM dual UNION ALL
   SELECT 20 x FROM dual UNION ALL
   SELECT 30 x FROM dual UNION ALL
   SELECT 40 x FROM dual UNION ALL
   SELECT 50 x FROM dual UNION ALL
   SELECT 60 x FROM dual UNION ALL
   SELECT 70 x FROM dual UNION ALL
   SELECT 80 x FROM dual UNION ALL
   SELECT 90 x FROM dual) tens,
  (SELECT 0 x FROM dual UNION ALL
   SELECT 100 x FROM dual UNION ALL
   SELECT 200 x FROM dual UNION ALL
   SELECT 300 x FROM dual) hundreds
```

Так как запрос не содержит инструкцию `WHERE`, будут сгенерированы все сочетания единиц, десятков и сотен, и сумма трех цифр каждой строки породит множество  $\{0, 1, 2, \dots, 399\}$ . Следующий запрос генерирует множество дней 2000 года, прибавляя каждый член множества к базовой дате и отбрасывая дни, которые попадают в 2001 год:

```
SELECT days.dt
FROM
```



```

(SELECT TO_DATE('01-JAN-2000', 'DD-MON-YYYY') +
    ones.x + tens.x + hundreds.x dt
FROM
    (SELECT 0 x FROM dual UNION ALL
     SELECT 1 x FROM dual UNION ALL
     SELECT 2 x FROM dual UNION ALL
     SELECT 3 x FROM dual UNION ALL
     SELECT 4 x FROM dual UNION ALL
     SELECT 5 x FROM dual UNION ALL
     SELECT 6 x FROM dual UNION ALL
     SELECT 7 x FROM dual UNION ALL
     SELECT 8 x FROM dual UNION ALL
     SELECT 9 x FROM dual) ones,
    (SELECT 0 x FROM dual UNION ALL
     SELECT 10 x FROM dual UNION ALL
     SELECT 20 x FROM dual UNION ALL
     SELECT 30 x FROM dual UNION ALL
     SELECT 40 x FROM dual UNION ALL
     SELECT 50 x FROM dual UNION ALL
     SELECT 60 x FROM dual UNION ALL
     SELECT 70 x FROM dual UNION ALL
     SELECT 80 x FROM dual UNION ALL
     SELECT 90 x FROM dual) tens,
    (SELECT 0 x FROM dual UNION ALL
     SELECT 100 x FROM dual UNION ALL
     SELECT 200 x FROM dual UNION ALL
     SELECT 300 x FROM dual) hundreds) days
WHERE days.dt < TO_DATE('01-JAN-2001', 'DD-MON-YYYY');
```

Так как 2000 год – високосный, результирующее множество будет содержать 366 строк, по одной для каждого дня. Этот запрос можно поместить в другое встроенное представление и использовать как управляющую таблицу для составления отчета. Вам решать, стоит ли на самом деле использовать этот метод в вашем коде; смысл примера был в том, чтобы помочь вам почувствовать вкус творчества.

## Преодоление ограничений SQL

Использование некоторых возможностей Oracle SQL может налагать ограничения на операторы SQL. Однако когда эти возможности применяются во встроенном представлении изолированно от остальной части запроса, ограничения можно преодолеть. В этом разделе вы узнаете о том, как встроенные представления борются с ограничениями, относящимися к иерархическим и обобщающим запросам.

### Иерархические запросы

Иерархические запросы делают возможным обход рекурсивных отношений. В качестве примера рекурсивного отношения рассмотрим таблицу *region*, которая хранит данные о регионах сбыта. Регионы орга-



низованы иерархически, и каждая запись таблицы регионов ссылается на регион, в котором она содержится:

```
SELECT * FROM region;
```

REGION_ID	REGION_NAME	SUPER_REGION_ID
1	North America	
2	Canada	1
3	United States	1
4	Mexico	1
5	New England	3
6	Mid-Atlantic	3
7	SouthEast US	3
8	SouthWest US	3
9	NorthWest US	3
10	Central US	3
11	Europe	
12	France	11
13	Germany	11
14	Spain	11

Каждая запись таблицы заказчиков ссылается на самый мелкий из подходящих регионов. Для определенного региона можно сформулировать запрос, который осуществит обход иерархической структуры вниз или вверх, используя инструкции `START WITH` и `CONNECT BY`:

```
SELECT region_id, name, super_region_id
FROM region
START WITH name = 'North America'
CONNECT BY PRIOR region_id = super_region_id;
```

REGION_ID	NAME	SUPER_REGION_ID
1	North America	
2	Canada	1
3	United States	1
5	New England	3
6	Mid-Atlantic	3
7	SouthEast US	3
8	SouthWest US	3
9	NorthWest US	3
10	Central US	3
4	Mexico	1

Запрос показывает иерархию регионов, начиная с Северной Америки (North America) и спускаясь вниз по дереву. Внимательно посмотрев на результаты, вы увидите, что регионы «Canada», «United States» и «Mexico» в поле `super_region_id` указывают на регион «North America». Все остальные строки указывают на регион «United States». Итак, выявлена трехуровневая иерархия с одним узлом на вершине, тремя узлами на втором уровне и шестью узлами на третьем уровне под уз-

лом «United States». Подробная информация об иерархических запросах будет приведена в главе 8.

Представьте, что вам нужно составить отчет с итогами продаж за 2001 год для каждого подрегиона Северной Америки. На иерархические запросы налагается ограничение: таблица, обход которой будет производиться, не может быть объединена с другими таблицами в том же запросе, так что, похоже, при помощи одного запроса сгенерировать отчет не удастся. Однако, используя встроенное представление, можно изолировать иерархический запрос к таблице регионов от таблиц `customer` и `cust_order`:

```
SELECT na_regions.name region_name,
       SUM(co.sale_price) total_sales
FROM   cust_order co, customer c,
       (SELECT region_id, name
        FROM region
        START WITH name = 'North America'
        CONNECT BY PRIOR region_id = super_region_id) na_regions
WHERE  co.cancelled_dt IS NULL
       AND co.order_dt >= TO_DATE('01-JAN-2001', 'DD-MON-YYYY')
       AND co.order_dt < TO_DATE('01-JAN-2002', 'DD-MON-YYYY')
       AND co.cust_nbr = c.cust_nbr
       AND c.region_id = na_regions.region_id
GROUP BY na_regions.name;
```

REGION_NAME	TOTAL_SALES
Central US	6238901
Mid-Atlantic	6307766
New England	6585641
NorthWest US	6739374
SouthEast US	6868495
SouthWest US	6854731

Несмотря на то что множество `na_regions` содержит регионы «North America» и «United States», записи заказчиков всегда указывают на самый мелкий из подходящих регионов, поэтому эти два региона не присутствуют в итоговом результирующем множестве.

Поместив иерархический запрос во встроенное представление, вы сможете временно сделать иерархическую структуру регионов «плоской», что позволит обойти ограничение на использование иерархических запросов, не прибегая к разбиению логики на отдельные фрагменты. В следующем разделе похожий подход будет применен для работы с обобщающими запросами.

## Обобщающие запросы

На запросы, выполняющие обобщение данных, налагается такое ограничение: все необобщаемые столбцы инструкции `SELECT` должны быть включены в инструкцию `GROUP BY`. Рассмотрим запрос, который сумми-

рует данные о продажах по заказчику и торговому представителю, а затем добавляет вспомогательные данные из таблиц заказчиков, регионов, служащих и подразделений:

```
SELECT c.name customer, r.name region,  
       e.fname || ' ' || e.lname salesperson, d.name department,  
       SUM(co.sale_price) total_sales  
FROM cust_order co, customer c, region r, employee e, department d  
WHERE co.cust_nbr = c.cust_nbr  
      AND c.region_id = r.region_id  
      AND co.sales_emp_id = e.emp_id  
      AND e.dept_id = d.dept_id  
      AND co.cancelled_dt IS NULL  
      AND co.order_dt >= TO_DATE('01-JAN-2001', 'DD-MON-YYYY')  
GROUP BY c.name, r.name, e.fname || ' ' || e.lname, d.name;
```

Поскольку каждый необобщаемый столбец инструкции SELECT должен быть включен в инструкцию GROUP BY, приходится выполнять сортировку по пяти столбцам, так как она необходима для создания групп. Но каждый заказчик находится в одном и только одном регионе, а каждый служащий работает в одном и ровно одном подразделении, поэтому на самом деле для вывода нужного результата необходима сортировка только по полям данных заказчика и служащего. То есть Oracle попусту теряет время, сортируя названия регионов и подразделений.

Отделив обобщение от вспомогательных таблиц, можно написать более эффективный и понятный запрос:

```
SELECT c.name customer, r.name region,  
       e.fname || ' ' || e.lname salesperson, d.name department,  
       cust_emp_orders.total total_sales  
FROM customer c, region r, employee e, department d,  
     (SELECT cust_nbr, sales_emp_id, SUM(sale_price) total  
      FROM cust_order  
      WHERE cancelled_dt IS NULL  
            AND order_dt >= TO_DATE('01-JAN-2001', 'DD-MON-YYYY'))  
GROUP BY cust_nbr, sales_emp_id) cust_emp_orders  
WHERE cust_emp_orders.cust_nbr = c.cust_nbr  
      AND c.region_id = r.region_id  
      AND cust_emp_orders.sales_emp_id = e.emp_id  
      AND e.dept_id = d.dept_id;
```

Так как таблица cust\_order содержит номер заказчика и идентификатор торгового представителя, можно выполнить суммирование по этим двум столбцам, не включая в него четыре остальные таблицы. При этом вы не просто сортируете меньшее количество столбцов, теперь сортируются только числовые поля (номер заказчика и идентификатор служащего), а не длинные строки (фамилия заказчика, название региона, фамилия служащего, название подразделения). Охватывающий запрос использует столбцы cust\_nbr и sales\_emp\_id встроен-



ного представления для объединения с таблицами заказчиков и служащих, которые, в свою очередь, объединяются с таблицами регионов и подразделений.

Выполняя обобщение внутри встроенного представления, мы обходим требование на включение всех необобщаемых столбцов в инструкцию GROUP BY. За счет удаления ненужных сортировок уменьшается время выполнения запроса; минимизировано количество объединений таблиц заказчиков, регионов, служащих и подразделений. В зависимости от объема данных в таблицах эти усовершенствования могут означать значительное увеличение производительности.

## Встроенные представления в операторах DML

Теперь, когда вы освоили встроенные представления, пришло время поговорить об еще одной их особенности: встроенные представления могут использоваться в операторах INSERT, UPDATE и DELETE. В большинстве случаев встроенное представление в операторе DML улучшает читабельность, но несколько увеличивает время выполнения запроса. Рассмотрим для начала простой оператор UPDATE и напомним эквивалентный оператор, использующий встроенное представление:

```
UPDATE cust_order co SET co.expected_ship_dt = co.expected_ship_dt + 7
WHERE co.cancelled_dt IS NULL AND co.ship_dt IS NULL
  AND EXISTS (SELECT 1 FROM line_item li, part p
              WHERE li.order_nbr = co.order_nbr
                 AND li.part_nbr = p.part_nbr
                 AND p.inventory_qty = 0);
```

Этот оператор использует условие EXISTS для нахождения заказов, содержащих детали, отсутствующие на складе. В следующей версии для определения того же самого множества заказов используется встроенное представление suspended\_orders:

```
UPDATE (SELECT co.expected_ship_dt exp_ship_dt
FROM cust_order co
WHERE co.cancelled_dt IS NULL AND co.ship_dt IS NULL
  AND EXISTS (SELECT 1 FROM line_item li, part p
              WHERE li.order_nbr = co.order_nbr
                 AND li.part_nbr = p.part_nbr
                 AND p.inventory_qty = 0)) suspended_orders
SET suspended_orders.exp_ship_dt = suspended_orders.exp_ship_dt + 7;
```

В первом варианте инструкция WHERE оператора UPDATE определяет набор строк для изменения, а во втором варианте этот же набор задает результирующее множество, возвращенное оператором SELECT. В остальном они идентичны. Чтобы улучшить оператор, встроенное представление должно делать нечто такое, чего простой оператор делать не умеет: объединять несколько таблиц. Сделаем попытку выполнить эту операцию, просто заменив подзапрос на объединение трех таблиц:

```

UPDATE (SELECT co.expected_ship_dt exp_ship_dt
FROM cust_order co, line_item li, part p
WHERE co.cancelled_dt IS NULL AND co.ship_dt IS NULL
AND co.order_nbr = li.order_nbr AND li.part_nbr = p.part_nbr
AND p.inventory_qty = 0) suspended_orders
SET suspended_orders.exp_ship_dt = suspended_orders.exp_ship_dt + 7;

```

Однако выполнение запроса приводит к ошибке:

ORA-01779: cannot modify a column which maps to a non key-preserved table

Как это часто бывает в жизни, ничего невозможно получить просто так. Чтобы использовать в операторе DML возможность объединения нескольких таблиц, мы должны следовать таким правилам:

- Только одна из объединяемых таблиц может быть изменена охватывающим оператором DML.
- Для того чтобы данные в таблице можно было изменять, ее ключ должен сохраняться в результирующем множестве встроеного представления.

Предыдущий оператор UPDATE пытается изменить только одну таблицу (cust\_order), но ключ (order\_nbr) не сохраняется в результирующем множестве, так как заказ содержит несколько строк. Другими словами, строки результирующего множества, образованного объединением трех таблиц, не могут быть однозначно идентифицированы по полю order\_nbr, поэтому изменить таблицу cust\_order, используя данное объединение трех таблиц, невозможно. Однако, используя то же самое объединение, можно изменить или удалить строки таблицы line\_item, так как ключ таблицы line\_item совпадает с ключом результирующего множества, возвращенного встроеным представлением (order\_nbr и part\_nbr). Следующий оператор удаляет строки из таблицы line\_item, используя встроеное представление, практически идентичное тому, которое не удалось выполнить в предыдущем UPDATE:

```

DELETE FROM (SELECT li.order_nbr order_nbr, li.part_nbr part_nbr
FROM cust_order co, line_item li, part p
WHERE co.cancelled_dt IS NULL AND co.ship_dt IS NULL
AND co.order_nbr = li.order_nbr AND li.part_nbr = p.part_nbr
AND p.inventory_qty = 0) suspended_orders;

```

Столбец (столбцы) инструкции SELECT встроеного представления фактически к делу не относятся. Так как таблица line\_item – это единственная из трех таблиц, перечисленных в инструкции FROM, которая сохраняет ключ, с ней и работает оператор DELETE. Хотя использование встроеного представления в операторе DELETE может улучшить производительность, несколько огорчает то, что сразу не очевидно, к какой таблице обращен оператор DELETE. При написании таких операторов разумно придерживаться следующей договоренности: всегда выбирать ключевые столбцы целевых таблиц.



## Ограничение доступа посредством WITH CHECK OPTION

Есть еще одна вещь, которую умеют делать встроенные представления и которая усовершенствует операторы DML – наложение ограничения на изменение строк и столбцов. Например, в большинстве компаний просмотр и изменение информации о заработной плате служащих разрешены только сотрудникам кадрового отдела. Ограничив ряд столбцов, видимых оператору DML, можно спрятать столбец salary:

```
UPDATE (SELECT emp_id, fname, lname, dept_id, manager_emp_id
FROM employee) emp
SET emp.manager_emp_id = 11
WHERE emp.dept_id = 4;
```

Такой оператор должен выполняться без проблем, а вот попытка добавить в инструкцию SET столбец зарплата должна вызвать ошибку:

```
UPDATE (SELECT emp_id, fname, lname, dept_id, manager_emp_id
FROM employee) emp
SET emp.manager_emp_id = 11, emp.salary = 1000000000
WHERE emp.dept_id = 4;

ORA-00904: invalid column name
```

Конечно, человек, пишущий оператор UPDATE, имеет полный доступ к таблице; смысл в том, чтобы запретить несанкционированные изменения со стороны пользователей. Это может оказаться полезным в многоуровневой модели, где интерфейсный уровень взаимодействует с уровнем бизнес-логики.

Этот механизм ограничивает доступ к отдельным столбцам, но не строкам. Чтобы ограничить множество строк, которые могут быть изменены оператором DML, можно добавить во встроенное представление инструкцию WHERE и указать WITH CHECK OPTION. Например, вы можете захотеть запретить пользователям изменять данные любого из сотрудников отдела кадров:

```
UPDATE (SELECT emp_id, fname, lname, dept_id, manager_emp_id
FROM employee
WHERE dept_id !=
(SELECT dept_id FROM department WHERE name = 'Human Resources'))
WITH CHECK OPTION) emp
SET emp.manager_emp_id = 11
WHERE emp.dept_id = 4;
```

Добавление во встроенное представление WITH CHECK OPTION заставляет оператор DML повиноваться инструкции WHERE встроенного представления. Попытка обновить или удалить данные сотрудников отдела кадров не удастся, но и исключения не возникнет (будет изменено или удалено 0 строк). Однако попытка добавить нового служащего в отдел кадров приведет к ошибке:

```
ORA-01402: view WITH CHECK OPTION where-clause violation
```



Соответственно, следующий оператор не удастся выполнить (ошибка ORA-01402), так как он пытается добавить служащего в отдел кадров:

```
INSERT INTO (SELECT emp_id, fname, lname, dept_id, manager_emp_id
FROM employee
WHERE dept_id !=
(SELECT dept_id FROM department
WHERE name = 'Human Resources')
WITH CHECK OPTION) emp
SELECT 99, 'Charles', 'Brown', d.dept_id, NULL
FROM department d
WHERE d.name = 'Human Resources';
```

## Изучаем пример подзапроса: N лучших работников

Некоторые запросы, которые легко описать словами, сложно сформулировать на языке SQL. В качестве примера рассмотрим запрос «Найти пять лучших продавцов». Сложность заключается в том, что данные таблицы сначала необходимо просуммировать, потом полученные обобщенные значения следует отсортировать и сравнить друг с другом для определения лучших или худших работников. В этом разделе вы узнаете, как в подобной ситуации можно использовать подзапросы. В конце раздела вы познакомитесь с ранжирующими функциями (новая функциональность Oracle SQL), которые были специально созданы для таких типов запросов.

## Посмотрим на данные

Рассмотрим задачу определения пяти лучших продавцов. Давайте будем использовать в качестве базы для расчетов размер выручки, которую каждый продавец принес за предыдущий год. Тогда первым делом необходимо вычислить сумму всех заказов, обслуженных каждым из продавцов в течение года. Рассмотрим 2001 год:

```
SELECT e.lname employee, SUM(co.sale_price) total_sales
FROM cust_order co, employee e
WHERE co.order_dt >= TO_DATE('01-JAN-2001', 'DD-MON-YYYY')
AND co.order_dt < TO_DATE('01-JAN-2002', 'DD-MON-YYYY')
AND co.ship_dt IS NOT NULL AND co.cancelled_dt IS NULL
AND co.sales_emp_id = e.emp_id
GROUP BY e.lname
ORDER BY 2 DESC;
```

EMPLOYEE	TOTAL_SALES
Blake	1927580
Houseman	1814327
Russell	1784596

Boorman	1768813
Isaacs	1761814
McGowan	1761814
Anderson	1757883
Evans	1737093
Fletcher	1735575
Dunn	1723305
Jacobs	1710831
Thomas	1695124
Powers	1688252
Walters	1672522
Fox	1645204
King	1625458
Nichols	1542152
Young	1516776
Grossman	1501039
Iverson	1468316
Freeman	1461898
Levitz	1458053
Peters	1443837
Jones	1392648

Оказывается, Isaacs и McGowan делят пятое место, что делает задачу еще более интересной.

## Ваше задание

Похоже, что начальник очень доволен результатами годовых продаж: он попросил вас, менеджера по информационным технологиям, проследить за тем, чтобы каждый из пяти лучших продавцов получил премию в размере 1% своих годовых продаж. «Нет проблем», – говорите вы, наспех составляете отчет, используя свое любимое средство – встроенное представление, и отправляете его начальнику:

```
SELECT e.ename employee, top5_emp_orders.tot_sales total_sales,
       ROUND(top5_emp_orders.tot_sales * 0.01) bonus
FROM
  (SELECT all_emp_orders.sales_emp_id emp_id,
         all_emp_orders.tot_sales tot_sales
   FROM
     (SELECT sales_emp_id, SUM(sale_price) tot_sales
      FROM cust_order
      WHERE order_dt >= TO_DATE('01-JAN-2001', 'DD-MON-YYYY')
        AND order_dt < TO_DATE('01-JAN-2002', 'DD-MON-YYYY')
        AND ship_dt IS NOT NULL AND cancelled_dt IS NULL
      GROUP BY sales_emp_id
      ORDER BY 2 DESC
     ) all_emp_orders
   WHERE ROWNUM <= 5
  ) top5_emp_orders, employee e
WHERE top5_emp_orders.emp_id = e.emp_id;
```

EMPLOYEE	TOTAL_SALES	BONUS
Blake	1927580	19276
Houseman	1814327	18143
Russell	1784596	17846
Boorman	1768813	17688
McGowan	1761814	17618

Крики, которые издает Isaacs, слышны по всей округе. Несколько обеспокоенный начальник просит вас переделать отчет. Исследовав свой запрос, вы быстро понимаете, в чем дело: встроенное представление суммирует данные о продажах и сортирует результаты, а охватывающий оператор «забирает» первые пять строк по порядку, а остальные отбрасывает. Так же мог пострадать и McGowan, но ему повезло, а Isaacs был удален из результирующего множества.

## Вторая попытка

Вы утешаете себя тем, что начальник получил именно то, о чем просил: пять лучших продавцов. Но все же понимаете, что часть вашей работы заключается в том, чтобы давать людям то, что им нужно, и это необязательно то, о чем они просят. Поэтому вы переформулируете задание босса следующим образом: выдать премию всем тем служащим, чей объем продаж попадает в пятерку лучших за последний год. Работаем поэтапно: сначала находим пятый лучший результат продаж за последний год, а затем ищем всех продавцов, объем продаж которых равен или превышает найденное значение.

```
SELECT e.ename employee, top5_emp_orders.tot_sales total_sales,
       ROUND(top5_emp_orders.tot_sales * 0.01) bonus
FROM employee e,
     (SELECT sales_emp_id, SUM(sale_price) tot_sales
      FROM cust_order
      WHERE order_dt >= TO_DATE('01-JAN-2001', 'DD-MON-YYYY')
        AND order_dt < TO_DATE('01-JAN-2002', 'DD-MON-YYYY')
        AND ship_dt IS NOT NULL AND cancelled_dt IS NULL
      GROUP BY sales_emp_id
      HAVING SUM(sale_price) IN
        (SELECT all_emp_orders.tot_sales
         FROM
           (SELECT SUM(sale_price) tot_sales
            FROM cust_order
            WHERE order_dt >= TO_DATE('01-JAN-2001', 'DD-MON-YYYY')
              AND order_dt < TO_DATE('01-JAN-2002', 'DD-MON-YYYY')
              AND ship_dt IS NOT NULL AND cancelled_dt IS NULL
            GROUP BY sales_emp_id
            ORDER BY 1 DESC
           ) all_emp_orders
         WHERE ROWNUM <= 5)
     ) top5_emp_orders
WHERE top5_emp_orders.sales_emp_id = e.emp_id
```



ORDER BY 2 DESC;

EMPLOYEE	TOTAL_SALES	BONUS
-----	-----	-----
Blake	1927580	19276
Houseman	1814327	18143
Russell	1784596	17846
Boorman	1768813	17688
McGowan	1761814	17618
Isaacs	1761814	17618

Таким образом, у вас получилось шесть продавцов из лучшей пятерки. Основное отличие первого подхода от второго заключается в добавлении во встроенное представление инструкции HAVING. Подзапрос инструкции HAVING возвращает пять лучших объемов продаж, а встроенное представление возвращает всех продавцов (возможно, их будет больше пяти), объем продаж которых присутствует во множестве, возвращенном подзапросом.

Вы довольны полученными результатами, но кое-что вам еще не нравится:

- Обобщение данных о продажах выполняется дважды.
- Запрос явно не может претендовать на титул «Самый элегантный запрос года».
- Вы можете поклясться, что где-то читали о новой возможности по обработке таких типов запросов...

И такая функциональность действительно есть. Начиная с версии Oracle 8.1.6 можно выполнять ранжирующие запросы при помощи функции RANK.

## Окончательный ответ

Появившаяся в версии Oracle 8.1.6 функция RANK специально создана для того, чтобы помочь в написании запросов для решения задач, подобных рассматриваемой. Будучи аналитической функцией (о них мы поговорим в главе 13), RANK может использоваться для назначения каждому элементу множества определенного ранга. Функция RANK понимает, что в множестве возможны равные значения, и для их компенсации она оставляет промежутки. Следующий запрос показывает, какие ранги будут присвоены всем продавцам. Обратите внимание на то, что поскольку пятое место в классификации делят две строки, за рангом 5 сразу следует ранг 7:

```
SELECT sales_emp_id, SUM(sale_price) tot_sales,
       RANK() OVER (ORDER BY SUM(sale_price) DESC) sales_rank
FROM   cust_order
WHERE  order_dt >= TO_DATE('01-JAN-2001','DD-MON-YYYY')
      AND order_dt < TO_DATE('01-JAN-2002','DD-MON-YYYY')
      AND ship_dt IS NOT NULL AND cancelled_dt IS NULL
```

GROUP BY sales\_emp\_id;

SALES_EMP_ID	TOT_SALES	SALES_RANK
11	1927580	1
24	1814327	2
34	1784596	3
18	1768813	4
25	1761814	5
26	1761814	5
30	1757883	7
21	1737093	8
19	1735575	9
20	1723305	10
27	1710831	11
14	1695124	12
15	1688252	13
22	1672522	14
29	1645204	15
28	1625456	16
31	1542152	17
23	1516776	18
13	1501039	19
32	1468316	20
12	1461898	21
17	1458053	22
33	1443837	23
16	1392648	24

Оставление промежутка после каждого совпадения рангов необходимо для корректной обработки таких запросов.<sup>1</sup> В табл. 5.1 перечислено количество строк, возвращаемых для запросов N лучших продавцов.

Таблица 5.1. Количество строк, возвращаемых для  $N=\{1,2,3,\dots,9\}$

N лучших продавцов	Возвращаемые строки
1	1
2	2
3	3
4	4
5	6
6	6
7	7
8	8
9	9

<sup>1</sup> Если вы не хотите, чтобы при ранжировании множества оставались промежутки, используйте функцию DENSE\_RANK.

Как видите, результирующие множества для версий данного запроса «пять лучших» и «шесть лучших» и данного конкретного множества данных будут совпадать.

Заклучив RANK-запрос во встроенное представление, вы сможете извлечь продавцов с рангом 5 и меньше и объединить результаты с таблицей служащих для получения итогового результирующего множества:

```
SELECT e.lname employee, top5_emp_orders.tot_sales total_sales,
       ROUND(top5_emp_orders.tot_sales * 0.01) bonus
FROM
  (SELECT all_emp_orders.sales_emp_id emp_id,
         all_emp_orders.tot_sales tot_sales
   FROM
     (SELECT sales_emp_id, SUM(sale_price) tot_sales,
            RANK() OVER (ORDER BY SUM(sale_price) DESC) sales_rank
      FROM cust_order
      WHERE order_dt >= TO_DATE('01-JAN-2001','DD-MON-YYYY')
            AND order_dt < TO_DATE('01-JAN-2002','DD-MON-YYYY')
            AND ship_dt IS NOT NULL AND cancelled_dt IS NULL
      GROUP BY sales_emp_id
     ) all_emp_orders
   WHERE all_emp_orders.sales_rank <= 5
  ) top5_emp_orders, employee e
WHERE top5_emp_orders.emp_id = e.emp_id
ORDER BY 2 DESC;
```

EMPLOYEE	TOTAL_SALES	BONUS
Blake	1927580	19276
Houseman	1814327	18143
Russell	1784596	17846
Boorman	1768813	17688
McGowan	1761814	17618
Isaacs	1761814	17618

Этот запрос может показаться знакомым, так как он почти идентичен вашей самой первой попытке, только для определения того, где провести черту между пятью лучшими продавцами и остальными сотрудниками, использован не псевдостолбец ROWNUM, а функция RANK.

Теперь вы довольны и запросом, и его результатами, и показываете полученные данные начальнику. «Хорошая работа, – говорит он. – Почему вы не дали премию себе самому? Кстати, можете получить премию, положенную Isaacs, – он уволился сегодня утром». Эти продавцы такие обидчивые...



# 6

## Обработка дат и времени

«Время не ждет» – говорится в известной поговорке. Разработчики баз данных постоянно имеют дело с данными, относящимися к датам и времени. Дата приема служащего на работу, дата зарплаты, дата выплаты кредита или арендной платы, срок окупаемости финансовых вложений, время и дата начала вашей новой страховки на автомобиль – все это примеры информации, с которой вы сталкиваетесь каждый день.

Потребность в эффективной обработке значений дат и времени становится критичной на рубеже веков, когда большинству из нас приходится изобретать способы корректного управления двузначными значениями годов, когда они переходят от 99 к 00, а затем к 01. В эпоху глобальной электронной коммерции понятие времени актуально как никогда ранее: торговля происходит двадцать четыре часа в сутки во всех временных зонах.

База данных должна эффективно и рационально организовывать хранение, извлечение и манипулирование следующих типов данных:

- Дата
- Время
- Интервалы дат и времени
- Часовые пояса

Обработка дат и времени в Oracle продуманна и эффективна. Oracle8i обеспечивает удобную работу с датами и временем. В Oracle9i вводится новый ряд возможностей, включая поддержку долей секунды, интервалов дат и времени и часовых поясов.

## Внутренний формат хранения даты

Тип данных `DATE` применяется в Oracle для хранения и даты и времени. Вне зависимости от применяемого пользователем формата даты, Oracle хранит даты в одном стандартном внутреннем формате. В рамках базы данных дата – это поле фиксированной длины 7 байт. Семь байт содержат следующие элементы информации:

1. Век
2. Год
3. Месяц
4. День
5. Час
6. Минута
7. Секунда

Обратите внимание, что хотя тип данных называется `DATE` (дата), он также хранит и время. При извлечении значения `DATE` из базы данных можно указать, какие компоненты необходимо отобразить (дату, время, дату и время и т. д.). Либо при выборке данных можно передать в программу (например, Java-программу) полное значение дата/время, а затем извлекать только нужные элементы.

## Вставка дат в БД и извлечение дат из БД

В реальном мире даты не всегда представляются в формате типа данных `DATE` Oracle. Постоянно будет возникать необходимость преобразования значений типа `DATE` в другие типы данных и наоборот. Это особенно важно при сопряжении базы данных Oracle с внешней системой, например, если данные получаются из внешней системы, в которой даты представлены символьными строками (или даже числами), или при отправке данных из базы Oracle в другие приложения, которые не поддерживают тип `DATE`. Также нужно преобразовывать значения `DATE` при отображении дат на экране или создании отчета.

Oracle предоставляет две чрезвычайно полезные функции преобразования дат:

- `TO_DATE`
- `TO_CHAR`

Как следует из их названий, функция `TO_DATE` используется для преобразования символьных или числовых данных в значение типа `DATE`, а функция `TO_CHAR` выполняет преобразование значения `DATE` в строку символов. Обсуждаемые далее в этом разделе форматы даты хорошо приспособлены для таких преобразований.

## TO\_DATE

TO\_DATE — это встроенная функция SQL, конвертирующая символьную строку в дату. На вход функции TO\_DATE может подаваться символьная строка, переменная PL/SQL или столбец базы данных типа CHAR или VARCHAR2.

Вызов TO\_DATE выглядит следующим образом:

```
TO_DATE(строка [, формат])
```

Перечислим элементы конструкции:

*строка*

Символьная строка, переменная PL/SQL или столбец базы данных, содержащий символьные (или числовые) данные, преобразуемые в дату.

*формат*

Задаёт формат преобразуемой строки. Формат должен представлять собой допустимую комбинацию элементов формата, представленных далее в этой главе (раздел «Форматы дат»).

Указание формата даты является необязательным. Если не задавать формат, то будет считаться, что строка имеет формат по умолчанию (определяемый параметром NLS\_DATE\_FORMAT).



С помощью функции TO\_DATE можно преобразовать число в формат DATE. Когда вы подаете на вход функции TO\_DATE число, Oracle неявно преобразует введенное число в строку, затем эта строка передается функции TO\_DATE.

### Использование для даты формата по умолчанию

Каждая база данных Oracle имеет формат даты по умолчанию. Если ваш администратор базы данных не определил ничего иного, то этот формат таков:

```
DD-MON-YY
```

При вызове функции TO\_DATE без явного указания формата даты Oracle считает, что строка ввода имеет формат даты по умолчанию. Следующий оператор INSERT преобразует строку в формате по умолчанию в значение типа DATE и вставляет его в таблицу EMPLOYEE:

```
INSERT INTO EMPLOYEE
(EMP_ID, FNAME, LNAME, DEPT_ID, MANAGER_EMP_ID, SALARY, HIRE_DATE)
VALUES
(2304, 'John', 'Smith', 20, 1258, 20000, TO_DATE('22-OCT-99'));

1 row created.

SELECT * FROM EMPLOYEE;
```



EMP_ID	FNAME	LNAME	DEPT_ID	MANAGER_EMP_ID	SALARY	HIRE_DATE
2304	John	Smith	20	1258	20000	22-OCT-99

Обратите внимание, что столбец HIRE\_DATE имеет тип DATE, и символьная строка '22-OCT-99' была преобразована в дату функцией TO\_DATE. В данном случае указание формата не требуется, так как вставляемая строка имеет формат даты по умолчанию. В действительности, если предлагаемая строка имеет формат даты по умолчанию, не нужна и сама функция TO\_DATE. Oracle автоматически выполняет неявное преобразование типов, как в приведенном ниже примере:

```
INSERT INTO EMPLOYEE
(EMP_ID, FNAME, LNAME, DEPT_ID, MANAGER_EMP_ID, SALARY, HIRE_DATE)
VALUES
(2304, 'John', 'Smith', 20, 1258, 20000, '22-OCT-99');
```

1 row created.

Но несмотря на то что Oracle производит неявное преобразование типов, рекомендуем всегда использовать явное преобразование, так как неявные преобразования не очевидны и могут привести к путанице. К тому же, если администратор базы данных изменит формат даты по умолчанию, неявные преобразования могут и не привести к желаемому результату.

## Указание формата даты

Если вы хотите указать формат даты, то существует как минимум два способа это сделать:

- Указать формат на уровне сеанса, тогда он будет применен ко всем неявным преобразованиям и ко всем преобразованиям в функции TO\_DATE, для которых явно не указан какой-то другой формат.
- Указать формат как параметр при вызове функции TO\_DATE.

В следующем примере показано задание формата даты для всего сеанса, а затем использование функции TO\_DATE для преобразования числа в дату:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'MMDDYY';
```

Session altered.

```
INSERT INTO EMPLOYEE
(EMP_ID, FNAME, LNAME, DEPT_ID, MANAGER_EMP_ID, SALARY, HIRE_DATE)
VALUES
(2304, 'John', 'Smith', 20, 1258, 20000, TO_DATE(102299));
```

1 row created.

Так как формат даты по умолчанию был изменен до выполнения преобразования, в функции TO\_DATE не требуется указывать формат даты в качестве входного параметра.



Можно передать функции TO\_DATE число, такое как 102299, понадеявшись на неявное преобразование числа в строку, выполняемое Oracle, но все же лучше передавать функции TO\_DATE строку.

Если попробовать выполнить такую вставку без указания формата даты по умолчанию, соответствующего формату даты в передаваемой строке, Oracle выдаст сообщение об ошибке при попытке преобразования данных:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YY';

Session altered.

INSERT INTO EMPLOYEE
(EMP_ID, FNAME, LNAME, DEPT_ID, MANAGER_EMP_ID, SALARY, HIRE_DATE)
VALUES
(2304, 'John', 'Smith', 20, 1258, 20000, TO_DATE('102299'));
(2304, 'John', 'Smith', 20, 1258, 20000, TO_DATE('102299'))

ERROR at line 4:
ORA-01861: literal does not match format string
```

Если вы не хотите изменять формат даты по умолчанию для всего сеанса, необходимо передать формат даты как второй параметр функции TO\_DATE:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YY';

Session altered.

INSERT INTO EMPLOYEE
(EMP_ID, FNAME, LNAME, DEPT_ID, MANAGER_EMP_ID, SALARY, HIRE_DATE)
VALUES
(2304, 'John', 'Smith', 20, 1258, 20000, TO_DATE('102299', 'MMDDYY'));

1 row created.

SELECT * FROM EMPLOYEE;
```

EMP_ID	FNAME	LNAME	DEPT_ID	MANAGER_EMP_ID	SALARY	HIRE_DATE
2304	John	Smith	20	1258	20000	22-OCT-99

Заметьте, что функция TO\_DATE интерпретирует строку '102299' как имеющую формат 'MMDDYY'. Отметьте также, что в результирующем множестве оператора SELECT дата выводится в формате по умолчанию, а не в том формате, в котором она была вставлена.

Давайте рассмотрим еще один пример, чтобы понять, как символьный столбец базы данных может быть преобразован в тип DATE. Пусть столбец REPORT\_ID таблицы REPORT хранит дату формирования отчета, и пусть эта дата имеет формат 'MMDDYYYY'. Используем функцию TO\_DATE для этого столбца, чтобы вывести дату формирования отчета:

```
SELECT SENT_TO, REPORT_ID, TO_DATE(REPORT_ID, 'MMDDYYYY') DATE_GENERATED
FROM REPORT;
```

SENT_TO	REPORT_I	DATE_GENE
-----	-----	-----
Manager	01011999	01-JAN-99
Director	01121999	12-JAN-99
Vice President	01231999	23-JAN-99

В этом примере функция `TO_DATE` преобразует в дату содержащиеся в столбце данные в формате `MMDDYYYY`. Затем для удобства отображения эта дата неявно конвертируется в символьную строку с использованием формата даты по умолчанию.

## TO\_CHAR

Функция `TO_CHAR` является обратной по отношению к `TO_DATE` и преобразует дату в символьную строку. Вызов `TO_CHAR` выглядит следующим образом:

```
TO_CHAR(дата [, формат])
```

Рассмотрим элементы конструкции:

*дата*

Переменная PL/SQL или столбец базы данных типа `DATE`.

*формат*

Указывает формат выводимой строки. Формат должен представлять собой допустимую комбинацию форматов, представленных далее в этой главе (раздел «Форматы дат»).

Указание формата даты является необязательным. Если не задавать формат, то дата выводится в формате по умолчанию (определяемом параметром `NLS_DATE_FORMAT`).

В следующем примере функция `TO_CHAR` применяется для преобразования вводимой даты в строку с использованием формата даты по умолчанию:

```
SELECT FNAME, TO_CHAR(HIRE_DATE) FROM EMPLOYEE;
```

FNAME	TO_CHAR(H
-----	-----
John	22-OCT-99

А вот как функция `TO_CHAR` применяется для преобразования вводимой даты в строку с явным указанием формата даты:

```
SELECT FNAME, TO_CHAR(HIRE_DATE, 'MM/DD/YY') FROM EMPLOYEE;
```

FNAME	TO_CHAR(
-----	-----
John	10/22/99



Бывают случаи, когда необходимо использовать функции `TO_CHAR` и `TO_DATE` вместе. Например, если вы хотите узнать, каким днем недели было 1 января 2000 года, можно выполнить такой запрос:

```
SELECT TO_CHAR(TO_DATE('01-JAN-2000', 'DD-MON-YYYY'), 'Day') FROM DUAL;

TO_CHAR(T
-----
Saturday
```

В данном примере сначала строка `'01-JAN-2000'` преобразуется в значение типа `DATE`, которое затем функция `TO_CHAR` преобразует в строку, представляющую день недели.

## Форматы дат

Дату можно выводить несколькими способами. Каждая страна, каждая отрасль промышленности имеет свой стандарт представления дат. Oracle предоставляет коды форматов дат, так что мы можем интерпретировать и выводить даты в самых разнообразных форматах.

Приведем простой пример вывода даты:

```
SELECT SYSDATE FROM DUAL;

SYSDATE
-----
03-OCT-01
```

По умолчанию дата выводится в формате `DD-MON-YY`. Этот формат использует два разряда для даты (отсутствующие знаки заполняются нулями слева), три разряда для месяца (три первые буквы английского названия месяца в верхнем регистре) и два разряда для года столетия (отсутствующие знаки заполняются нулями слева). Формат даты по умолчанию для базы данных определяется инициализационным параметром `NLS_DATE_FORMAT`. Для изменения формата даты по умолчанию для экземпляра и для сеанса используйте команды `ALTER SYSTEM` и `ALTER SESSION` соответственно. Давайте рассмотрим еще один пример, чтобы показать, как выводить дату в формате, отличном от используемого по умолчанию:

```
SELECT TO_CHAR(SYSDATE, 'MM/DD/YYYY') FROM DUAL;

TO_CHAR(SY
-----
10/03/2001
```

Дата преобразуется в формат `'MM/DD/YYYY'` при помощи функции `TO_CHAR`. Существует множество способов представления дат. В разных странах, на разных производствах, в разных приложениях могут использоваться разные форматы дат. В табл. 6.1 приведен список форматов дат. В большинстве примеров этой таблицы используется дата 3 октября 2001 года, время 15:34:48. При обсуждении дат до нашей эры в ка-

честве примера используется 2015 год до нашей эры. Для рассмотрения времени до полудня используется дата 3 октября 2001 года, время 11:00:00.

Таблица 6.1. Коды форматов дат Oracle

Компонент	Варианты	Пример	
		Формат	Дата
Пунктуация	-, : , . , ^	DD-MON-YY DD MM YYYY DD «of» Month	03-OCT-01 03 10 2001 03 of October
День	DD (день месяца)	MM/DD/YY	10/03/01
	DDD (день года)	DDD/YY	276/01
	D (день недели)	D MM/YY	4 10/01
	DAY (название дня)	DAY MM/YY	WEDNESDAY 10/01
	day (название дня в нижнем регистре)	day MM/YY	wednesday 10/01
	Day (название дня с заглавной буквы)	Day MM/YY	Wednesday 10/01
	DY (аббревиатура названия дня)	DY MM/YY	WED 10/01
	Dy (аббревиатура названия дня)	Dy MM/YY	Wed 10/01
	Месяц	MM (двузначный номер месяца)	MM/DD/YY 10/03/01
	MONTH (название месяца в верхнем регистре)	MONTH YY	OCTOBER 01
Месяц	Month (название месяца с заглавной буквы)	Month YY	October 01
	MON (аббревиатура названия месяца)	MON YY	OCT 01
	Mon (аббревиатура названия месяца с заглавной буквы)	Mon YY	Oct 01
	RM (месяц римскими цифрами)	DD-RM-YY	03-X-01
	Год	Y (последняя цифра года)	MM Y 10 1
Год	YY (две последние цифры года)	MM YY	10 01

Компонент	Варианты	Пример	
		Формат	Дата
Год	YYY (три последние цифры года)	MM YYY	10 001
	YYYY (четыре цифры года)	MM YYYY	10 2001
	Y,YYY (год с запятой)	MM Y,YYY	10 2,001
	SYYYY (четыре цифры года со знаком «-» для дат до нашей эры)	SYYYY	-2105
	YEAR (год прописью)	MM YEAR	10 TWO THOUSAND ONE
	Year (год прописью с заглавной буквы)	MM Year	10 Two Thousand One
	RR (год, зависящий от текущего года)	DD-MON-RR	03-OCT-01
	RRRR (год, зависящий от текущего года)	DD-MON-RRRR	03-OCT-2001
	I (последняя цифра года в стандарте ISO)	MM I	10 1
	IY (две последние цифры года в стандарте ISO)	MM IY	10 01
	IYY (три последние цифры года в стандарте ISO)	MM IYY	10 001
	IYYY (четыре цифры года в стандарте ISO)	MM IYYY	10 2001
Век	CC (век)	CC	21
	SCC (век со знаком «-» для дат до нашей эры)	SCC	-22
Неделя	W (неделя месяца)	W	1
	WW (неделя года)	WW	40
	IW (неделя года в стандарте ISO)	IW	40
Квартал	Q (квартал года)	Q	4
Час	HH (час дня 1-12)	HH	03
	HH12 (час дня 1-12)	HH	03
	HH24 (час дня 0-23)	HH24	15
Минута	MI (минута часа 0-59)	MI	34



Таблица 6.1 (продолжение)

Компонент	Варианты	Пример	
		Формат	Дата
Секунда	SS (секунда минуты 0–59)	SS	48
	SSSSS (секунды после полудня)	SSSSS	42098
AM/PM	AM (указатель времени до полудня)	HH:MI AM	11:00 AM
	A.M. (указатель времени до полудня с точками)	HH:MI A.M.	11:00 A.M.
	PM (указатель времени после полудня)	HH:MI PM	03:34 PM
	P.M. (указатель времени после полудня с точками)	HH:MI P.M.	03:34 P.M.
AD/BC	AD (указатель даты нашей эры)	YY AD	01 AD
	A.D. (указатель даты нашей эры с точками)	YY A.D.	01 A.D.
	BC (указатель даты до нашей эры)	YY BC	05 BC
	B.C. (указатель даты до нашей эры с точками)	YY B.C.	05 B.C.
Юлианский календарь	J (количество дней после 1 января 4712 года до нашей эры)	J	2452186
Суффикс	TH (порядковые числительные)	DDTH	03RD
	SP (число прописью)	MMSP	TEN
	SPTH (порядковое числительное прописью)	DDSPTH	THIRD
	THSP (порядковое числительное прописью)	DDTHSP	THIRD

## Индикаторы AD/BC

Oracle предоставляет два формата, AD и BC, для характеристики года (и еще два с точками – A.D., B.C.). Однако оба они интерпретируются одинаково, и применение любого из них приводит к одному и тому же результату. Если в запросе использован формат BC, а дата, к которой применяется формат, относится к нашей эре (AD), то Oracle достаточно разумен, чтобы вывести AD вместо BC, и наоборот. Например:

## Минуты: MI или MM

Многие новички в SQL полагают, что раз HH представляет часы, а SS – секунды, MM должно обозначать минуты, и пишут для вывода текущего времени такие запросы:

```
SELECT TO_CHAR(SYSDATE, 'HH:MM:SS') FROM DUAL;
```

```
TO_CHAR(  
-----  
02:10:32
```

Однако результат неверен. MM представляет месяцы, а не минуты. Формат для минут – MI. Помните, что для вывода в составе даты минут необходимо использовать MI, а не MM. Правильный запрос выглядит следующим образом:

```
SELECT TO_CHAR(SYSDATE, 'HH:MI:SS') FROM DUAL;
```

```
TO_CHAR(  
-----  
02:57:21
```

Если в коде вместо формата MI указано MM, отладить приложение чрезвычайно трудно.

```
SELECT TO_CHAR(SYSDATE, 'YYYY AD'),  
       TO_CHAR(SYSDATE, 'YYYY BC') FROM DUAL;
```

```
TO_CHAR( TO_CHAR(  
-----  
2001 AD  2001 AD
```

```
SELECT TO_CHAR(ADD_MONTHS(SYSDATE, -50000), 'YYYY AD'),  
       TO_CHAR(ADD_MONTHS(SYSDATE, -50000), 'YYYY BC') FROM DUAL;
```

```
TO_CHAR( TO_CHAR(  
-----  
2165 BC  2165 BC
```

В первом примере, несмотря на то что дата введена в формате BC, выводится она как 2001 AD. А во втором примере с указанием формата AD вводится дата, обозначающая 50 000 месяцев до нашей эры, и на выходе она выглядит как BC.

## Индикаторы AM/PM

Индикаторы AM/PM (а также A.M. и P.M.) ведут себя в точности так же, как индикаторы AD/BC. Если вы используете в запросе формат AM, но при этом время, к которому применяется этот формат, оказывается

временем после полудня (PM), Oracle оказывается достаточно умен для того, чтобы вывести PM вместо AM, и наоборот. Например (см. врезку):

```
SELECT TO_CHAR(SYSDATE, 'HH:MI:SS AM'),
       TO_CHAR(SYSDATE, 'HH:MI:SS PM'),
       TO_CHAR(SYSDATE - 8/24, 'HH:MI:SS AM'),
       TO_CHAR(SYSDATE - 8/24, 'HH:MI:SS PM')
FROM DUAL;

TO_CHAR(SYS TO_CHAR(SYS TO_CHAR(SYS TO_CHAR(SYS
-----
06:58:07 PM 06:58:07 PM 10:58:07 AM 10:58:07 AM
```

## Чувствительность форматов к регистру

Есть форматы дат чувствительные к регистру, а есть нечувствительные. Форматы, представляющие числа, не чувствительны к регистру. Например:

```
SELECT TO_CHAR(SYSDATE, 'HH:MI') UPPER,
       TO_CHAR(SYSDATE, 'hh:mi') LOWER,
       TO_CHAR(SYSDATE, 'Hh:mI') MIXED
FROM DUAL;

UPPER LOWER MIXED
-----
03:17 03:17 03:17
```

Как видите, неважно, какой регистр используется для формата – выводится одно и то же. Это справедливо и для других форматов, представляющих числа, например DD, MM, YY и т. п.

Форматы дат, представляющие текстовые компоненты даты, чувствительны к смене регистра. Например, формат «DAY» отличается от «day». Для определения регистра вывода при использовании текстового формата даты применяются следующие правила:

- Если первый символ формата введен в нижнем регистре, то независимо от регистра остальных символов формата данные выводятся в нижнем регистре.

```
SELECT TO_CHAR(SYSDATE, 'month'),
       TO_CHAR(SYSDATE, 'mONTH'),
       TO_CHAR(SYSDATE, 'moNTh')
FROM DUAL;

TO_CHAR(S TO_CHAR(S TO_CHAR(S
-----
october  october  october
```

- Если первый и второй символы формата введены в верхнем регистре, то независимо от регистра остальных символов формата данные выводятся в верхнем регистре.



```

SELECT TO_CHAR(SYSDATE, 'Month'),
       TO_CHAR(SYSDATE, 'MONTH')
FROM DUAL;

TO_CHAR(S TO_CHAR(S
-----
OCTOBER    OCTOBER

```

- Если первый символ формата введен в верхнем регистре, а второй символ – в нижнем регистре, то независимо от регистра остальных символов формата первый символ выводимых данных будет в верхнем регистре, а остальные – в нижнем.

```

SELECT TO_CHAR(SYSDATE, 'Month'), TO_CHAR(SYSDATE, 'month')
FROM DUAL;

TO_CHAR(S TO_CHAR(S
-----
October    October

```

Эти правила относятся ко всем текстовым элементам, которые применяются для представления названий месяцев, дней недели и т. д.

## Двузначный год

Несмотря на то что Oracle во внутреннем представлении дат хранит век, он позволяет использовать двузначные значения года. Поэтому важно понимать, что происходит с веком, когда вы используете двузначное представление года. Oracle поддерживает два двузначных формата года: YY и RR.

В формате года YY две первые цифры трактуются в соответствии с текущей датой:

```

ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YY';

Session altered.

SELECT SYSDATE, TO_CHAR(SYSDATE, 'DD-MON-YYYY') FROM DUAL;

SYSDATE    TO_CHAR(SYSDATE, 'DD-MON-YYYY')
-----
06-OCT-01  06-OCT-2001

SELECT TO_CHAR(TO_DATE('10-DEC-99'), 'DD-MON-YYYY'),
       TO_CHAR(TO_DATE('10-DEC-01'), 'DD-MON-YYYY') FROM DUAL;

TO_CHAR(TO_ TO_CHAR(TO_
-----
10-DEC-2099 10-DEC-2001

```

Так как текущей датой было 6 октября 2001 года, то первые две цифры значения года были интерпретированы как 20.

В формате года RR две первые цифры года определяются по двум последним цифрам текущего года и двум последним цифрам указанного года с помощью следующих правил:

- Если указанный год меньше 50 и последние две цифры текущего года также меньше 50, то две первые цифры возвращаемой даты совпадают с двумя первыми цифрами текущей даты.
- Если указанный год меньше 50, а последние две цифры текущего года больше или равны 50, то две первые цифры возвращаемой даты на единицу больше, чем две первые цифры текущей даты.
- Если указанный год больше 50, а последние две цифры текущего года меньше 50, то две первые цифры возвращаемой даты на единицу меньше, чем две первые цифры текущей даты.
- Если указанный год больше 50 и последние две цифры текущего года также больше 50, то две первые цифры возвращаемой даты совпадают с двумя первыми цифрами текущей даты.

Продemonстрируем вышеприведенные правила на примерах:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-RR';

Session altered.

SELECT SYSDATE, TO_CHAR(SYSDATE, 'DD-MON-YYYY') FROM DUAL;

SYSDATE      TO_CHAR(SYS
-----
06-OCT-01    06-OCT-2001

SELECT TO_CHAR(TO_DATE('10-DEC-99'), 'DD-MON-YYYY'),
       TO_CHAR(TO_DATE('10-DEC-01'), 'DD-MON-YYYY') FROM DUAL;

TO_CHAR(TO_  TO_CHAR(TO_
-----
10-DEC-1999 10-DEC-2001
```

Команда ALTER SESSION устанавливает формат даты по умолчанию в DD-MON-RR. Следующий оператор SELECT использует функцию SYSDATE для вывода текущей даты на момент выполнения. В последнем операторе SELECT используется формат даты RR (оба вызова TO\_DATE полагаются на установленный ранее формат даты по умолчанию). Заметьте, что формат DD-MON-RR интерпретирует 10-DEC-99 как 10-DEC-1999, а 10-DEC-01 — как 10-DEC-2001. Вспомните о правилах преобразования.

Формат года RRRR (четыре R) позволяет вводить как двузначные, так и четырехзначные значения для года. Если вводится четырехзначное значение, Oracle ведет себя так, как если бы использовался формат года YYYY. Если же вводится двузначный год, Oracle поступает так, как если бы форматом года был RR. Формат RRRR используется редко. Большинство SQL-программистов предпочитает применять YYYY или явно указывать RR.



## Литералы дат

Литерал DATE используется в стандарте ANSI для представления дат в следующем формате:

```
DATE 'YYYY-MM-DD'
```

Обратите внимание на то, что литерал дат ANSI не содержит информацию о времени. Также нет возможности указать формат даты. При необходимости указать литерал, используя синтаксис ANSI, придется применять только формат YYYY-MM-DD. Приведем пример использования литерала DATE в операторе SQL:

```
INSERT INTO EMPLOYEE  
(EMP_ID, FNAME, LNAME, DEPT_ID, MANAGER_EMP_ID, SALARY, HIRE_DATE)  
VALUES  
(2304, 'John', 'Smith', 20, 1258, 20000, DATE '1999-10-22');
```

1 row created.

```
SELECT * FROM EMPLOYEE;
```

EMP_ID	FNAME	LNAME	DEPT_ID	MANAGER_EMP_ID	SALARY	HIRE_DATE
2304	John	Smith	20	1258	20000	22-OCT-99

В данном примере литерал DATE '1999-10-22' интерпретируется как 22-OCT-99.

## Стандарты ISO

Стандарт ISO определяет начало первой недели года по тому, принадлежит ли большинство ее дней старому или же новому году. Если 1 января – это понедельник, вторник, среда или четверг, то 1 января относится к 1-й неделе нового ISO-года. Первый день года ISO – это или 1 января (если это понедельник), или предшествующий понедельник (который на самом деле относится к предыдущему календарному году). Например, если 1 января приходится на вторник, то первый день года в стандарте ISO – это понедельник, 31 декабря предыдущего календарного года.

Если первое января выпадает на пятницу, субботу или воскресенье, то первое января, согласно стандарту ISO, будет относиться к предыдущему году. Тогда первым днем нового года считается первый понедельник после 1 января. Например, если 1 января – пятница, то первым днем нового года в стандарте ISO будет понедельник, 3 января.

Для тех, кому нужно работать с датами в формате ISO, Oracle предоставляет форматы, интерпретирующие годы ISO не так, как календарные:

**IW**

Представляет неделю года в стандарте ISO.



*I, IY, IYY и IYYY*

Представляет год ISO.

Далее форматы недель и годов стандарта ISO рассмотрены на примерах.

## Недели в стандарте ISO

В стандарте ISO недели года считаются не так, как обычные календарные недели. В календаре первая неделя года начинается 1 января. 1 января – это первый день первой недели. В стандарте ISO неделя всегда начинается с понедельника и заканчивается воскресеньем. Поэтому первым днем первой недели года считается понедельник, ближайший к 1 января. Этот день может быть на пару дней раньше или позже 1 января.

Формат **WW** возвращает неделю года в терминах обычного календаря, а формат **IW** – в терминах стандарта ISO. Так как в 2001 году 1 января было понедельником, оно рассматривалось как начало первой недели года и для обычного календаря, и для ISO. Поэтому при вычислении номера недели для любой даты 2001 года результаты использования обычного календаря и стандарта ISO будут совпадать. Например:

```
SELECT TO_CHAR(TO_DATE('10-DEC-01'), 'WW'),  
       TO_CHAR(TO_DATE('10-DEC-01'), 'IW')  
FROM DUAL;  
  
TO TO  
-- --  
50 50
```

Однако 1999 год начался не с понедельника. Поэтому для некоторых дат номер недели в стандарте ISO может отличаться от номера недели для стандартного календаря, например:

```
SELECT TO_CHAR(TO_DATE('10-DEC-99'), 'WW'),  
       TO_CHAR(TO_DATE('10-DEC-99'), 'IW')  
FROM DUAL;  
  
TO TO  
-- --  
50 49
```

В стандарте ISO может случиться так, что в году будет 53 недели:

```
SELECT TO_CHAR(TO_DATE('01-JAN-99'), 'IW'), TO_CHAR(TO_DATE('01-JAN-  
99'), 'Day')  
FROM DUAL;  
  
TO TO_CHAR(T  
-----  
53 Friday
```

Как видите, ISO рассматривает 1 января 1999 года принадлежащим 53-й неделе 1998 года, так как этот день – четверг. Первая неделя 1999 года в стандарте ISO начнется в следующий понедельник, 4 января.

## Год в стандарте ISO

Форматы года I, IY, IYY и IYYYY представляют год в стандарте ISO. IYYYY соответствует четырехзначному обозначению года в стандарте ISO, IYY представляет три последние цифры года ISO, IY – две последние цифры года ISO и Y – последнюю цифру года ISO. Помните, что началом года в стандарте ISO не обязательно будет 1 января. В следующем примере возвращаются календарный год и год ISO для 1 января 1999 года:

```
SELECT TO_CHAR(TO_DATE('01-JAN-99'), 'YYYY'),  
       TO_CHAR(TO_DATE('01-JAN-99'), 'YYYY') FROM DUAL;  
  
TO_C TO_C  
----  
1998 1999
```

Обратите внимание на то, что 1 января 1999 года относится к 1999 календарному году, а стандарт ISO воспринимает эту дату как 1998 год, так как 1 января 1999 – это четверг, день в конце недели, поэтому эта неделя считается принадлежащей предыдущему, 1998 году ISO. Рассмотрим противоположную ситуацию:

```
SELECT TO_CHAR(TO_DATE('31-DEC-90'), 'YYYY'),  
       TO_CHAR(TO_DATE('31-DEC-90'), 'YYYY') FROM DUAL;  
  
TO_C TO_C  
----  
1991 1990
```

Теперь на календаре 1990 год, но стандарт ISO трактует 31 декабря 1990 года как день, относящийся к 1991 году ISO. Дело в том, что 1 января 1991 года – вторник, день в начале недели, поэтому неделя интерпретируется как первая неделя следующего года.

## Работа с датами

Арифметические операции с датами являются важным аспектом нашей с вами повседневной жизни. Вы узнаете возраст человека, вычитая год его рождения из нынешнего. Вы определяете дату окончания гарантии, прибавляя гарантийный срок к дате покупки. Истечение срока действия водительских прав, начисление банковских процентов – все это и многое другое зависит от арифметических операций над датами. Для любой базы данных чрезвычайно важна поддержка общепринятых арифметических операций с датами.

Oracle предоставляет множество полезных арифметических функций. Можно не только складывать и вычитать даты, существует и ряд дру-

гих функций для обработки значений дат, о которых будет рассказано в данном разделе. В табл. 6.2 приведены различные арифметические функции Oracle SQL, работающие с датами.

Таблица 6.2. Функции, работающие с датами

Функция	Описание
ADD_MONTHS	Добавляет к дате месяцы
LAST_DAY	Определяет последний день месяца
MONTHS_BETWEEN	Вычисляет количество месяцев между двумя указанными датами
NEW_TIME	Переводит время в новую временную зону
NEXT_DAY	Возвращает дату дня недели, следующего за указанным
ROUND	Округляет дату/время до указанного элемента
SYSDATE	Возвращает текущую дату и время
TO_CHAR	Преобразовывает даты в строки
TO_DATE	Преобразовывает строки и числа в даты
TRUNC	Сокращает значение даты/времени до указанного элемента

## Сложение

Сложение двух дат не имеет смысла. Но вы можете прибавлять к дате дни, месяцы, годы, часы, минуты и секунды, чтобы получить будущую дату и время. Оператор «+» позволяет прибавлять к дате числа. Считается, что единицей значения, прибавляемого к дате, является день. Следовательно, для вывода завтрашней даты можно прибавить единицу к SYSDATE:

```
SELECT SYSDATE, SYSDATE+1 FROM DUAL;
```

```
SYSDATE    SYSDATE+1
```

```
-----  
05-OCT-01 06-OCT-01
```

Каждый раз, когда вы прибавляете к дате число, Oracle считает, что оно представляет количество дней. Поэтому при желании добавить к дате несколько дней (недель, месяцев, лет) необходимо сначала выполнить умножение на соответствующий коэффициент. Например, чтобы добавить к сегодняшней дате неделю, прибавьте к SYSDATE 7 (7 дней в неделе умножить на 1 неделю):

```
SELECT SYSDATE+7 FROM DUAL;
```

```
SYSDATE+7
```

```
-----  
12-OCT-01
```



Аналогично, если вы хотите добавить к дате часть дня (часы, минуты, секунды), сначала необходимо преобразовать эти часы, минуты или секунды в дробное число дней. Для этого выполняем деление на соответствующий коэффициент. Например, чтобы прибавить к текущему времени текущего дня 20 минут, необходимо выполнить следующую операцию сложения (20 минут/1440 минут в сутках):

```
SELECT TO_CHAR(SYSDATE, 'DD-MON-YY HH:MI:SS'),
       TO_CHAR(SYSDATE+(20/1440), 'DD-MON-YY HH:MI:SS')
FROM DUAL;

TO_CHAR(SYSDATE, 'D TO_CHAR(SYSDATE+(2
-----
05-OCT-01 01:22:03 05-OCT-01 01:42:03
```

Месяцы добавлять не так легко, как недели, так как в них содержится разное количество дней: в некоторых 30, в некоторых 31, 28 и иногда 29. Чтобы добавить к дате месяц, необходимо знать, сколько в данном календарном месяце дней. Поэтому добавление к дате месяцев путем преобразования их в дни требует непростых вычислений, которые могут повлечь за собой ошибки. К счастью, Oracle делает всю работу за нас, предоставляя встроенную функцию SQL, добавляющую к датам месяцы. Функция `ADD_MONTHS` вызывается следующим образом:

`ADD_MONTHS (дата, число)`

Перечислим элементы синтаксиса:

*дата*

Указывает столбец базы данных, определенный как тип `DATE`, или строку с датой в формате даты по умолчанию.

*число*

Указывает количество месяцев, добавляемых к введенной дате.

В следующем примере производится расчет даты для производимой раз в полгода аттестации служащего. Функция `ADD_MONTHS` используется для добавления 6 месяцев к дате приема сотрудника на работу, получаемой из поля `HIRE_DATE`:

```
SELECT FNAME, HIRE_DATE, ADD_MONTHS(HIRE_DATE, 6) REVIEW_DATE FROM EMPLOYEE;

FNAME                HIRE_DATE REVIEW_DA
-----
John                 22-OCT-99 22-APR-00
```

Заметьте, что в данном примере и дата, являющаяся входным параметром функции, и результирующая дата выпадают на 22 число. Так не случилось бы, если бы вы просто добавили 180 дней к дате приема на работу. Функция `ADD_MONTHS` умеет и еще кое-что. Добавим 6 месяцев к 31 декабря 1999 года:

```
SELECT ADD_MONTHS('31-DEC-99',6) FROM DUAL;
```

```
ADD_MONTH
```

```
-----
```

```
30-JUN-00
```

Функция `ADD_MONTHS` настолько умна, что знает, что в результате добавления 6 месяцев к 31 декабря должен получиться последний день июня. А так как в июне 30 дней, возвращается 30 июня 2000 года.

## Вычитание

Сложение, умножение и деление дат не имеет смысла, а вот вычитание — это чрезвычайно полезная и очень распространенная операция. Оператор «-» позволяет вычитать из даты число или другую дату.

Результатом вычитания одной даты из другой является количество дней, разделяющих эти даты. Вычитание числа из даты возвращает дату, имевшую место указанное количество дней назад.

В следующем примере выводится время выполнения заказов, получаемое путем вычитания даты получения заказа (`ORDER_DT`) из предполагаемой даты отгрузки (`EXPECTED_SHIP_DT`):

```
SELECT ORDER_NBR, EXPECTED_SHIP_DT - ORDER_DT LEAD_TIME  
FROM CUST_ORDER;
```

```
ORDER_NBR  LEAD_TIME
```

```
-----
```

```
1001      1
```

```
1000      5
```

```
1002     13
```

```
1003     10
```

```
1004      9
```

```
1005      2
```

```
1006      6
```

```
1007      2
```

```
1008      2
```

```
1009      4
```

```
1012      1
```

```
1011      5
```

```
1015     13
```

```
1017     10
```

```
1019      9
```

```
1021      2
```

```
1023      6
```

```
1025      2
```

```
1027      2
```

```
1029      4
```

Как уже говорилось, из даты можно вычитать не только дату, но и число. Например, вычтя 1 из SYSDATE, вы получите вчерашний день, вычтя 7 — тот же день на прошлой неделе:

```
SELECT SYSDATE, SYSDATE - 1, SYSDATE - 7 FROM DUAL;
```

```
SYSDATE    SYSDATE-1 SYSDATE-7
```

```
-----  
05-OCT-01 04-OCT-01 28-SEP-01
```

В отличие от сложения (ADD\_MONTHS), Oracle не предоставляет функцию для вычитания (SUBTRACT\_MONTHS). Чтобы вычесть из даты месяцы, используйте функцию ADD\_MONTHS, передавая в качестве второго параметра отрицательное число:

```
SELECT SYSDATE, ADD_MONTHS(SYSDATE, -6) FROM DUAL;
```

```
SYSDATE    ADD_MONTH
```

```
-----  
05-OCT-01 05-APR-01
```

При вычитании одной даты из другой получается количество дней между ними. Иногда необходимо узнать количество месяцев, разделяющих две даты. Вычитая из SYSDATE дату приема сотрудника на работу, содержащуюся в столбце HIRE\_DATE, вы получаете количество отработанных им в компании дней:

```
SELECT SYSDATE-HIRE_DATE FROM EMPLOYEE;
```

```
SYSDATE-HIRE_DATE
```

```
-----  
714.0786
```

В данном случае, как и во многих других, интереснее было бы получить количество месяцев, а не дней. Вы знаете, что разделив количество дней между двумя датами на 30, вы не получите точное количество месяцев. Поэтому Oracle предоставляет встроенную функцию SQL MONTHS\_BETWEEN для вычисления количества месяцев между двумя датами, которая вызывается следующим образом:

```
MONTHS_BETWEEN (дата1, дата2)
```

Перечислим элементы конструкции:

*дата1*

Указывает конец рассматриваемого периода времени. Значение типа DATE или строка в формате даты по умолчанию.

*дата2*

Указывает начало рассматриваемого периода времени. Как и дата1, это должно быть значение типа DATE или строка в формате даты по умолчанию.



Функция MONTHS\_BETWEEN вычитает значение дата2 из значения дата1. Если дата2 хронологически больше, чем дата1, то MONTHS\_BETWEEN возвращает отрицательное значение. Приведем два примера вызова функции MONTHS\_BETWEEN, в которых использованы одни и те же даты, но в разном порядке:

```
SELECT MONTHS_BETWEEN(SYSDATE, HIRE_DATE)
       MONTHS_BETWEEN(HIRE_DATE, SYSDATE)
FROM EMPLOYEE;

MONTHS_BETWEEN(SYSDATE, HIRE_DATE) MONTHS_BETWEEN(HIRE_DATE, SYSDATE)
-----
23.4542111                        -23.454218
```

Функции YEARS\_BETWEEN не существует. Чтобы найти количество лет, прошедших между двумя датами, можно выполнить вычитание, найти количество дней между двумя датами и разделить его на 365 или же использовать функцию MONTHS\_BETWEEN для вычисления количества месяцев и разделить полученный результат на 12. Не все годы содержат по 365 дней, в некоторых 366 дней. Поэтому некорректно делить количество дней на 365 для получения количества годов. Зато любой год, будь то високосный или обычный, состоит из 12 месяцев. Следовательно, наиболее точным способом вычисления количества лет между двумя датами является применение функции MONTHS\_BETWEEN для вычисления количества месяцев и последующее деление полученного результата на 12.

## Последний день месяца

Oracle содержит встроенную функцию, возвращающую последний день месяца. Функция LAST\_DAY вызывается следующим образом:

```
LAST_DAY (дата)
```

Элемент синтаксиса:

*дата*

Указывает значение типа DATE или строку в формате даты по умолчанию.

Функция LAST\_DAY возвращает последний день месяца, содержащего переданную в качестве параметра дату. Например, чтобы найти последний день текущего месяца, можно использовать такой оператор SQL:

```
SELECT LAST_DAY(SYSDATE) "Next Payment Date" FROM DUAL;

Next Paym
-----
31-OCT-01
```

В некоторых случаях полезно иметь возможность вычислить первый день заданного месяца, и было бы хорошо, если бы в Oracle была функция FIRST\_DAY. Первый день месяца для указанной даты можно вычислить при помощи следующего выражения:

```
ADD_MONTHS((LAST_DAY(date)+1), -1)
```

В этом способе сначала вычисляется последний день месяца для указанной даты. К нему добавляется единица для получения первого дня следующего месяца, а затем используется функция ADD\_MONTHS с аргументом -1 для возвращения к началу того месяца, с которого были начаты вычисления. В результате получаем первый день месяца для указанной в аргументе даты. Возможны другие способы решения данной задачи; мы рассмотрели лишь один из них. Плюсы данного подхода в том, что он сохраняет составляющую рассматриваемой даты, содержащую информацию о времени.

## Следующий день

Oracle предоставляет встроенную функцию для получения даты следующего указанного дня недели. Функция NEXT\_DAY вызывается следующим образом:

```
NEXT_DAY (дата, строка)
```

Элементы синтаксиса:

*дата*

Значение типа DATE или строка в формате даты по умолчанию.

*строка*

Название дня недели.

Чтобы узнать дату следующего четверга, используем следующий оператор SQL:

```
SELECT NEXT_DAY(SYSDATE, 'Friday') "Vacation Start Date" FROM DUAL;
```

```
Vacation
```

```
-----
```

```
12-OCT-01
```

Если введенная строка – не день недели, будет выдано сообщение об ошибке:

```
SELECT NEXT_DAY(SYSDATE, 'ABCD') FROM DUAL;
```

```
SELECT NEXT_DAY(SYSDATE, 'ABCD') FROM DUAL
```

```
ERROR at line 1:
```

```
ORA-01846: not a valid day of the week
```

## Округление и усечение дат

Округление и усечение дат – процесс, напоминающий округление и усечение чисел, только более запутанный, так как значения данных типа DATE содержат информацию не только о дате, но и о времени. Для округления даты/времени до указанного элемента используйте функцию ROUND. Для усечения даты/времени до указанного элемента используйте функцию TRUNC. Синтаксис вызова этих двух функций:

```
ROUND(дата [, формат])  
TRUNC(дата [, формат])
```

Элементы синтаксиса:

*дата*

Значение типа DATE.

*формат*

Указывает элемент даты для округления или усечения.

Возвращаемое значение зависит от заданного формата, являющегося необязательным параметром. Если в вызове ROUND не указан формат, функция возвращает дату, округляя входное значение до ближайшего дня. Если формат не указан в вызове TRUNC, функция возвращает дату, отсекая дробную часть дня.

Если функции ROUND и TRUNC используются для округления до ближайшего дня или для усечения даты, то поля времени возвращаемого значения устанавливаются на начало дня, то есть 12:00:00 AM (00:00:00 в формате HH24), например:

```
SELECT TO_CHAR(SYSDATE, 'DD-MON-YY HH:MI:SS AM'),  
       TO_CHAR(ROUND(SYSDATE), 'DD-MON-YY HH:MI:SS AM'),  
       TO_CHAR(TRUNC(SYSDATE), 'DD-MON-YY HH:MI:SS AM')  
FROM DUAL;  
  
TO_CHAR(SYSDATE, 'DD-M TO_CHAR(ROUND(SYSDATE TO_CHAR(TRUNC(SYSDATE  
-----  
06-OCT-01 07:35:48 AM 06-OCT-01 12:00:00 AM 06-OCT-01 12:00:00 AM
```

Обратите внимание на то, что входное время (SYSDATE) – это время до полудня, поэтому результаты работы функций ROUND и TRUNC совпадают. Но если бы на вход функций подавалось время после полудня, результаты функций ROUND и TRUNC отличались бы друг от друга. Рассмотрим пример:

```
SELECT TO_CHAR(SYSDATE, 'DD-MON-YY HH:MI:SS AM'),  
       TO_CHAR(ROUND(SYSDATE), 'DD-MON-YY HH:MI:SS AM'),  
       TO_CHAR(TRUNC(SYSDATE), 'DD-MON-YY HH:MI:SS AM')  
FROM DUAL;  
  
TO_CHAR(SYSDATE, 'DD-M TO_CHAR(ROUND(SYSDATE TO_CHAR(TRUNC(SYSDATE  
-----  
06-OCT-01 05:35:48 PM 07-OCT-01 12:00:00 AM 06-OCT-01 12:00:00 AM
```



Так как рассматривается время после полудня, ROUND возвращает начало следующего дня, а TRUNC продолжает возвращать начало того же дня. Все аналогично округлению и усечению чисел.

Если при вызове функций ROUND и TRUNC указать формат, ситуация немного усложнится, но принцип округления и усечения останется неизменным. Разница в том, что теперь округление и усечение используют указанный формат.

Например, если указать формат YYYY, то усечение будет производиться относительно года, то есть если введенная дата относится к первой половине года (до 1 июля), то ROUND и TRUNC будут возвращать первый день года. Если же дата относится ко второй половине года, ROUND будет возвращать первый день следующего года, а TRUNC будет возвращать первый день года введенной даты, например:

```
SELECT TO_CHAR(SYSDATE-180, 'DD-MON-YYYY HH24:MI:SS'),
       TO_CHAR(ROUND(SYSDATE-180, 'YYYY'), 'DD-MON-YYYY HH24:MI:SS'),
       TO_CHAR(TRUNC(SYSDATE-180, 'YYYY'), 'DD-MON-YYYY HH24:MI:SS')
FROM DUAL;
```

```
TO_CHAR(SYSDATE-180, TO_CHAR(ROUND(SYSDAT TO_CHAR(TRUNC(SYSDAT
-----
09-APR-2001 20:58:33 01-JAN-2001 00:00:00 01-JAN-2001 00:00:00
```

```
SELECT TO_CHAR(SYSDATE, 'DD-MON-YYYY HH24:MI:SS'),
       TO_CHAR(ROUND(SYSDATE, 'YYYY'), 'DD-MON-YYYY HH24:MI:SS'),
       TO_CHAR(TRUNC(SYSDATE, 'YYYY'), 'DD-MON-YYYY HH24:MI:SS')
FROM DUAL;
```

```
TO_CHAR(SYSDATE, 'DD- TO_CHAR(ROUND(SYSDAT TO_CHAR(TRUNC(SYSDAT
-----
06-OCT-2001 20:58:49 01-JAN-2002 00:00:00 01-JAN-2001 00:00:00
```

Аналогично можно округлять и усекать дату до месяца, недели, часа, минуты и т. д., задавая соответствующий формат. Форматы, которые можно указывать в функциях ROUND и TRUNC (и их значения), приведены в табл. 6.3.

Таблица 6.3. Форматы дат, используемые в функциях ROUND и TRUNC

Единица округления	Формат	Комментарий
Век	CC	TRUNC возвращает первый день века.
	SCC	Если дата относится к первой половине века (до 1 января xx51 года), ROUND возвращает первый день века; иначе ROUND возвращает первый день следующего века.

Таблица 6.3 (продолжение)

Единица округления	Формат	Комментарий
Год	YYYY	TRUNC возвращает первый день года.
	YYYY	Если дата относится к первой половине года (до 1 июля), ROUND возвращает первый день года; иначе ROUND возвращает первый день следующего года.
	YEAR	
	SYEAR	
	YYY	
	YY	
	Y	
ISO	IYYY	TRUNC возвращает первый день года в стандарте ISO.
	IYY	Если дата относится к первой половине года ISO, ROUND возвращает первый день года ISO; иначе ROUND возвращает первый день следующего года в стандарте ISO.
	IY	
	I	
Квартал	Q	TRUNC возвращает первый день квартала.
		Если дата относится к первой половине квартала (до 16 числа второго месяца квартала), ROUND возвращает первый день квартала; иначе ROUND возвращает первый день следующего квартала.
Месяц	MONTH	TRUNC возвращает первый день месяца.
	MON	Если дата относится к первой половине месяца (до 16 числа), ROUND возвращает первый день месяца; иначе ROUND возвращает первый день следующего месяца.
	MM	
	RM	
Неделя	WW	TRUNC возвращает первый день недели.
		Если дата относится к первой половине недели (отсчитываются от первого дня года), ROUND возвращает первый день недели; иначе ROUND возвращает первый день следующей недели.
Неделя ISO	IW	TRUNC возвращает первый день недели в стандарте ISO.
		Если дата относится к первой половине недели (отсчитываются от первого дня года ISO), ROUND возвращает первый день недели; иначе ROUND возвращает первый день следующей недели.
Неделя	W	TRUNC возвращает первый день недели.
		Если дата относится к первой половине недели (отсчитываются от первого дня месяца), ROUND возвращает первый день недели; иначе ROUND возвращает первый день следующей недели.
День	DDD	TRUNC возвращает начало дня.
	DD	Если дата относится к первой половине дня (до полудня), ROUND возвращает начало дня; иначе ROUND возвращает начало следующего дня.
	J	

Единица округления	Формат	Комментарий
День недели	DAY	TRUNC возвращает первый день недели.
	DY	Если дата относится к первой половине недели (отсчитываются от первого дня месяца), ROUND возвращает первый день недели; иначе ROUND возвращает первый день следующей недели.
	D	
Час	HH	TRUNC возвращает начало часа.
	HH12	Если дата относится к первой половине часа (до 00:30), ROUND возвращает начало часа; иначе ROUND возвращает начало следующего часа.
	HH24	
Минута	MI	TRUNC возвращает начало минуты. Если дата относится к первой половине минуты (до 00:00:30), возвращает начало минуты; иначе ROUND возвращает начало следующей минуты.

## NEW\_TIME

Пусть вы работаете в офисе в Нью-Йорке и хотите запланировать видеоконференцию с клиентом, находящимся в Лос-Анджелесе. Если вы не подумаете о разнице во времени между двумя городами, то можете, например, назначить встречу на 9 часов утра по вашему времени. На самом деле вы, конечно, знаете, что это неудачное время для тех, кто действительно хочет заключить сделку, ведь клиента еще просто не будет на работе (9:00 утра в Нью-Йорке – это 6:00 утра в Лос-Анджелесе). Если при работе с базой данных приходится иметь дело с часовыми поясами, на помощь приходит встроенная функция Oracle `NEW_TIME`. Она преобразует дату и время указанного часового пояса в дату и время другого часового пояса. Формат вызова `NEW_TIME` таков:

```
NEW_TIME (дата, входной_часовой_пояс, выходной_часовой_пояс)
```

Элементы синтаксиса:

*дата*

Литерал, переменная PL/SQL или столбец базы данных типа `DATE`.

*входной\_часовой\_пояс*

Указывает название исходного часового пояса (строка).

*выходной\_часовой\_пояс*

Указывает название производного часового пояса (строка).

Например, чтобы узнать время в Лос-Анджелесе, зная, что в Нью-Йорке девять часов утра, используем следующий запрос:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YY HH:MI:SS AM';
```

```
Session altered.
```



```
SELECT NEW_TIME('11-NOV-01 09:00:00 AM', 'EST', 'PST') FROM DUAL;
```

```
NEW_TIME('11-NOV-0109
```

```
-----  
11-NOV-01 06:00:00 AM
```

В данном примере EST и PST соответствуют стандартному времени восточного побережья и стандартному тихоокеанскому времени.

## Выбор данных по диапазону дат

Бывают случаи, когда необходимо выбрать данные из таблицы для какого-то диапазона дат. Пусть, например, нужно вывести все заказы, размещенные в определенный день, например 22 мая 2001 года. Вероятно, первым, что придет вам в голову, будет запрос:

```
SELECT * FROM CUST_ORDER  
WHERE ORDER_DT = '22-MAY-01';
```

```
no rows selected
```

Никакого вывода. Удивлены? Вы знаете, что 22 мая заказы были, но запрос не вернул ни одной строки. Причина в том, что ORDER\_DT – это столбец типа DATE и содержит как дату, так и время. А вот литерал '22-MAY-01' не содержит никакой информации о времени. Если в литерале не указано время, оно интерпретируется как начало дня, то есть 12:00:00 до полудня (или 00:00:00 в двадцатичетырехчасовом формате). В столбце ORDER\_DT таблицы CUST\_ORDER время получения заказа отлично от 12:00:00 А.М. Поэтому корректный запрос для вывода заказов, размещенных 22 мая 2001 года, должен быть таким:

```
SELECT * FROM CUST_ORDER  
WHERE ORDER_DT BETWEEN TO_DATE('22-MAY-01 00:00:00', 'DD-MON-YY HH24:MI:SS')  
AND TO_DATE('22-MAY-01 23:59:59', 'DD-MON-YY HH24:MI:SS');
```

ORDER_NBR	CUST	SALES_EMP	SALE_PRICE	ORDER_DT	EXPECTED	CANCELLED	SHIP STATUS
1001	1	3	99	22-MAY-01	23-MAY-01		DELIVERED
1005	8	3	99	22-MAY-01	24-MAY-01		DELIVERED
1021	8	7	99	22-MAY-01	24-MAY-01		DELIVERED

Запрос рассматривает один день как диапазон от 00:00:00 22 мая 2001 года до 23:59:59 22 мая 2001 года. Поэтому такой запрос возвращает все заказы, размещенные в течение всего дня 22 мая 2001 года.

Еще одним способом решения данной проблемы – игнорирования данных о времени в столбце типа DATE – является усечение даты и сравнение усеченного результата с входным литералом:

```
SELECT * FROM CUST_ORDER  
WHERE TRUNC(ORDER_DT) = '22-MAY-01';
```

```
ORDER_NBR CUST SALES_EMP SALE_PRICE ORDER_DT EXPECTED CANCELLED SHIP STATUS
```

1001	1	3	99 22-MAY-01 23-MAY-01	DELIVERED
1005	8	3	99 22-MAY-01 24-MAY-01	DELIVERED
1021	8	7	99 22-MAY-01 24-MAY-01	DELIVERED

Функция TRUNC устанавливает время на начало дня. Поэтому сравнение с литералом '22-MAY-01' возвращает ожидаемый результат. Тот же результат можно получить, преобразовав данные из столбца ORDER\_DT в символьную строку в формате, соответствующем формату входной даты:

```
SELECT * FROM CUST_ORDER
WHERE TO_CHAR(ORDER_DT, 'DD-MON-YY') = '22-MAY-01';
```

Минусы использования функций TRUNC и TO\_CHAR в том, что получившийся запрос не может воспользоваться индексом для столбца ORDER\_DT, что может значительно ухудшить производительность. С другой стороны, решение, применяющее диапазон дат, требует более сложного кода, но зато не исключает возможности использования индекса для столбца.



Версии Oracle8i и выше поддерживают индексы на основе функций, которые будучи корректно созданными позволяют использовать индексы, даже если функции применяются к столбцам.

Способ, представленный в данном разделе, может применяться для выбора данных для любого заданного диапазона дат, в том числе выходящего за рамки одного дня.

## Создание сводной таблицы дат

В некоторых типах запросов бывает полезно иметь таблицу, содержащую по одной строке для каждой даты некоторого периода. Например, вы можете захотеть иметь по строке для каждой даты текущего года. Для создания такой таблицы можно использовать функцию TRUNC вместе с кодом PL/SQL:

```
CREATE TABLE DATES_OF_YEAR (ONE_DAY DATE);

Table created.

DECLARE
  I NUMBER;
  START_DAY DATE := TRUNC(SYSDATE, 'YY');
BEGIN
  FOR I IN 0 .. (TRUNC(ADD_MONTHS(SYSDATE, 12), 'YY') - 1) -
    (TRUNC(SYSDATE, 'YY'))
  LOOP
    INSERT INTO DATES_OF_YEAR VALUES (START_DAY+I);
  END LOOP;
END;
/
```

PL/SQL procedure successfully completed.

```
SELECT COUNT(*) FROM DATES_OF_YEAR;
```

```
COUNT(*)
```

```
-----  
365
```

Таблица DATES\_OF\_YEAR заполнена 365 днями 2001 года. Теперь вы можете использовать эту таблицу для формирования различных полезных списков дат.

Пусть, например, вы получаете заработную плату дважды в месяц: 15 числа и в последний день месяца. Используем следующий запрос к таблице DATES\_OF\_YEAR для вывода списка всех дней выдачи зарплаты в 2001 году.

```
SELECT ONE_DAY PAYDAY FROM DATES_OF_YEAR  
WHERE TO_CHAR(ONE_DAY, 'DD') = '15'  
OR ONE_DAY = LAST_DAY(ONE_DAY);
```

```
PAYDAY
```

```
-----
```

```
15-JAN-01
```

```
31-JAN-01
```

```
15-FEB-01
```

```
28-FEB-01
```

```
15-MAR-01
```

```
31-MAR-01
```

```
15-APR-01
```

```
30-APR-01
```

```
15-MAY-01
```

```
31-MAY-01
```

```
15-JUN-01
```

```
30-JUN-01
```

```
15-JUL-01
```

```
31-JUL-01
```

```
15-AUG-01
```

```
31-AUG-01
```

```
15-SEP-01
```

```
30-SEP-01
```

```
15-OCT-01
```

```
31-OCT-01
```

```
15-NOV-01
```

```
30-NOV-01
```

```
15-DEC-01
```

```
31-DEC-01
```

```
24 rows selected.
```

Часто бывает так, что в какой-то правительственной организации вам говорят, что работа с документами займет «х» дней. Говоря нечто подобное, они всегда подразумевают количество рабочих дней. То есть



чтобы вычислить предполагаемую дату завершения работ, необходимо отсчитать «х» дней от текущей даты, при этом пропуская субботы и воскресенья. Понятно, что использовать просто арифметическую операцию нельзя, вычитание не исключит из рассмотрения выходные дни. Используем таблицу DATES\_OF\_YEAR:

```
SELECT COUNT(*) FROM DATES_OF_YEAR
WHERE RTRIM(TO_CHAR(ONE_DAY, 'DAY')) NOT IN ('SATURDAY', 'SUNDAY')
AND ONE_DAY BETWEEN '&d1' AND '&d2';
```

Enter value for d1: 18-FEB-01

Enter value for d2: 15-MAR-01

old 3: AND ONE\_DAY BETWEEN '&d1' AND '&d2'

new 3: AND ONE\_DAY BETWEEN '18-FEB-01' AND '15-MAR-01'

```
COUNT(*)
```

```
-----
19
```

Данный запрос подсчитывает количество дней между двумя введенными датами, не учитывая субботы и воскресенья. Функция TO\_CHAR с форматом 'DAY' преобразует каждую рассматриваемую дату (из таблицы DATES\_OF\_YEAR) в название дня недели, а оператор NOT IN исключает субботы и воскресенья. Обратите внимание на применение функции RTRIM к функции TO\_CHAR. Дело в том, что TO\_CHAR выводит название дня недели в виде строки длиной девять символов с пробелами-заполнителями справа. Функция RTRIM удаляет такие пробелы.

В диапазоне между двумя датами могут встречаться и праздничные дни, а ни один из рассмотренных в разделе запросов не учитывает такую возможность. Чтобы обрабатывать праздничные дни, необходимо создать еще одну таблицу (например, HOLIDAYS), в которой будут собраны все праздники года. Тогда вы сможете изменить предыдущий запрос, исключив даты, присутствующие в таблице HOLIDAYS.

## Суммирование по элементам дата/время

Предположим, вы хотите напечатать квартальную сводку по всем заказам. Вам нужно вывести общее количество заказов и общую отпускную цену за каждый квартал. Таблица заказов выглядит следующим образом:

```
SELECT * FROM CUST_ORDER;
```

ORDER_NBR	CUST	SALES	PRICE	ORDER_DT	EXPECTED_	CANCELLED	SHIP	STATUS
1001	1	3	99	22-MAY-01	23-MAY-01			DELIVERED
1000	1	4		19-JAN-01	24-JAN-01	21-JAN-01		CANCELLED
1002	5	6		12-JUL-01	25-JUL-01	14-JUL-01		CANCELLED
1003	4	5	56	16-NOV-01	26-NOV-01			DELIVERED
1004	4	4	34	18-JAN-01	27-JAN-01			PENDING

1005	8	3	99	22-MAY-01	24-MAY-01		DELIVERED
1006	1	8		22-JUL-01	28-JUL-01	24-JUL-01	CANCELLED
1007	5	1	25	20-NOV-01	22-NOV-01		PENDING
1008	5	1	25	21-JAN-01	23-JAN-01		PENDING
1009	1	5	56	18-MAY-01	22-MAY-01		DELIVERED
1012	1	2	99	22-JAN-01	23-JAN-01		DELIVERED
1011	1	3		19-NOV-01	24-NOV-01	21-NOV-01	CANCELLED
1015	5	3		12-NOV-01	25-NOV-01	14-NOV-01	CANCELLED
1017	4	1	56	16-MAY-01	26-MAY-01		DELIVERED
1019	4	9	34	18-NOV-01	27-NOV-01		PENDING
1021	8	7	99	22-MAY-01	24-MAY-01		DELIVERED
1023	1	1		22-NOV-01	28-NOV-01	24-NOV-01	CANCELLED
1025	5	3	25	20-MAY-01	22-MAY-01		PENDING
1027	5	1	25	21-NOV-01	23-NOV-01		PENDING
1029	1	5	56	18-MAY-01	22-MAY-01		DELIVERED

20 rows selected.

Столбца квартала в таблице CUST\_ORDER нет. Для его формирования вам необходимо поработать со столбцом ORDER\_DT. Следующий оператор SQL выполняет требуемое действие, используя функцию TO\_CHAR и формат даты. Заметьте, что функция TO\_CHAR встречается не только в списке SELECT, но и в инструкции GROUP BY для поквартальной группировки результатов.

```
SELECT 'Q' || TO_CHAR(ORDER_DT, 'Q') QUARTER, COUNT(*), SUM(NVL(SALE_PRICE,0))
FROM CUST_ORDER
GROUP BY 'Q' || TO_CHAR(ORDER_DT, 'Q');
```

QU	COUNT(*)	SUM(NVL(SALE_PRICE,0))
Q1	4	158
Q2	7	490
Q3	2	0
Q4	7	140

Используя тот же прием, можно просуммировать данные для недель, месяцев, годов, часов, минут и любой другой выбранной единицы даты/времени.

## Новые возможности Oracle9i по обработке даты и времени

Oracle9i предоставляет новые возможности по работе с данными, относящимися к дате и времени, которые обеспечивают поддержку работы с:

- Часовыми поясами
- Датами и временем с долями секунд
- Интервалами дат и времени

В данном разделе будет представлено описание нововведений и их применение.

## Часовые пояса

Электронная коммерция в Интернете выходит за рамки географических зон и часовых поясов. Oracle способствует глобализации электронной коммерции, обеспечивая поддержку работы с часовыми поясами. В Oracle9i база данных и сеанс могут быть сопоставлены с часовыми поясами. Наличие часового пояса базы данных и сеанса позволяет пользователям в географически удаленных регионах обмениваться с базой данных информацией о датах и времени, не думая о разнице их локального времени и времени места размещения сервера.

### Часовой пояс базы данных

При создании базы данных можно указать для нее часовой пояс. После того как база данных создана, можно изменить часовой пояс, используя команду ALTER DATABASE. Команды CREATE DATABASE и ALTER DATABASE могут содержать необязательную инструкцию SET TIME\_ZONE. Можно задать часовой пояс одним из двух способов:

- Указывая смещение относительно UTC (Universal Coordinated Time – всеобщее скоординированное время).
- Указывая региональный часовой пояс.

Смещение относительно UTC задается в часах и минутах со знаком «+» или «-». Каждый региональный часовой пояс имеет свое название. Например, EST – это название восточного стандартного времени (Eastern Standard Time). Региональное название может использоваться и для указания часового пояса базы данных.



Время UTC ранее называлось GMT (Greenwich Mean Time – Гринвичское время).

Синтаксис инструкции SET TIME\_ZONE таков:

```
SET TIME_ZONE = '+ | - HH:MI' | 'код_часового_пояса'
```

В следующих примерах инструкция используется для указания часового пояса базы данных:

```
CREATE DATABASE ... SET TIME_ZONE = '-05:00';
```

```
ALTER DATABASE ... SET TIME_ZONE = 'EST';
```

В обоих примерах часовой пояс устанавливается в Eastern Standard Time. В первом примере задано смещение (-05:00) относительно UTC, а во втором использовано название (EST).





Если часовой пояс базы данных не установлен явно, Oracle по умолчанию считает его совпадающим с часовым поясом операционной системы. Если часовой пояс операционной системы не относится к приемлемым для Oracle часовым поясам, используется UTC.

## Часовой пояс сеанса

Каждый сеанс также может иметь часовой пояс. Часовой пояс сеанса устанавливается при помощи инструкции `ALTER SESSION SET TIME_ZONE`. Синтаксис инструкции `SET TIME_ZONE` оператора `ALTER SESSION` такой же, как и в операторах `CREATE DATABASE` и `ALTER DATABASE`.

В следующем примере показаны два способа установки часового пояса сеанса в стандартное тихоокеанское время (Pacific Standard Time):

```
ALTER SESSION SET TIME_ZONE = '-08:00';
```

```
ALTER SESSION SET TIME_ZONE = 'PST';
```

Чтобы установить часовой пояс сеанса в часовой пояс локальной операционной системы (например, часовой пояс компьютера, инициировавшего сеанс работы удаленного пользователя), можно использовать в инструкции `SET TIME_ZONE` ключевое слово `LOCAL`, например:

```
ALTER SESSION SET TIME_ZONE = LOCAL;
```

Чтобы установить часовой пояс сеанса равным часовому поясу сервера базы данных, используйте в инструкции `SET TIME_ZONE` ключевое слово `DBTIMEZONE`, например:

```
ALTER SESSION SET TIME_ZONE = DBTIMEZONE;
```

Далее мы подробнее поговорим о ключевом слове `DBTIMEZONE`.



Если часовой пояс сеанса не был установлен явно, Oracle использует часовой пояс локальной операционной системы. Если часовой пояс операционной системы не относится к приемлемым для Oracle часовым поясам, используется UTC.

## Дата и время с долями секунды

Чтобы обеспечить поддержку работы с датами и временем, содержащими доли секунд, Oracle9i вводит новые типы данных:

- `TIMESTAMP`
- `TIMESTAMP WITH TIMEZONE`
- `TIMESTAMP WITH LOCAL TIMEZONE`

Эти типы данных делают возможной обработку значений времени с точностью до долей секунды и в разных часовых поясах. Упомянутые типы будут рассмотрены в следующих разделах.

## TIMESTAMP

Тип данных **TIMESTAMP** является расширением типа **DATE** для работы с более точными значениями времени. **TIMESTAMP** включает в себя все компоненты типа **DATE** (век, год, месяц, день, час, минута, секунда) и, кроме того, доли секунды. Тип данных **TIMESTAMP** задается следующим образом:

```
TIMESTAMP [ (точность долей секунды) ]
```

В скобках указывается точность долей секунды – вы можете задать целое значение от 0 до 9. Точность 9 означает, что возможны 9 знаков после десятичной точки. Из нотации вы видите, что задание точности необязательно. Если точность не задана, по умолчанию она считается равной 6, то есть **TIMESTAMP** – это то же самое, что и **TIMESTAMP(6)**.

Создадим таблицу со столбцом типа **TIMESTAMP**:

```
CREATE TABLE TRANSACTION (
  TRANSACTION_ID NUMBER(10),
  TRANSACTION_TIMESTAMP TIMESTAMP,
  STATUS VARCHAR2(12));
```

Table created.

DESC TRANSACTION

Name	Null?	Type
TRANSACTION_ID		NUMBER(10)
TRANSACTION_TIMESTAMP		TIMESTAMP(6)
STATUS		VARCHAR2(12)

Заметьте, что хотя в качестве типа столбца **TRANSACTION\_TIMESTAMP** указан просто **TIMESTAMP**, при описании таблицы он выводится как **TIMESTAMP(6)**. Чтобы вставить данные в этот столбец, можно использовать литерал **TIMESTAMP** в следующем формате:

```
TIMESTAMP 'YYYY-MM-DD HH:MI:SS.xxxxxxxxxx'
```

Литерал **TIMESTAMP** может иметь до 9 разрядов долей секунды. Доли секунды являются необязательным компонентом, в то время как дата и время обязательны и должны быть указаны в соответствующем формате. Приведем пример вставки данных в таблицу, содержащую столбец типа **TIMESTAMP**:

```
INSERT INTO TRANSACTION
VALUES (1001, TIMESTAMP '1998-12-31 08:23:46.368', 'OPEN');
```

1 row created.

```
SELECT * FROM TRANSACTION;
```

TRANSACTION_ID	TRANSACTION_TIMESTAMP	STATUS
1001	31-DEC-98 08.23.46.368000 AM	OPEN

## TIMESTAMP WITH TIME ZONE

Тип данных `TIMESTAMP WITH TIME ZONE` является дальнейшим расширением типа `TIMESTAMP`, включающим смещение для часового пояса. Тип `TIMESTAMP WITH TIME ZONE` задается следующим образом:

```
TIMESTAMP [ (точность долей секунды) ] WITH TIME ZONE
```

Точность долей секунды определяется так же, как и для типа данных `TIMESTAMP`. Смещение часового пояса – это разница времени (в часах и минутах) между локальным временем и GMT (или, что то же самое, UTC). Вы указываете смещения при вводе сохраняемых в столбце значений, а база данных запоминает их, так что в дальнейшем эти значения могут быть преобразованы для любого необходимого пользователю часового пояса.

Создадим таблицу со столбцом `TIMESTAMP`:

```
CREATE TABLE TRANSACTION_TIME_ZONE (  
  TRANSACTION_ID NUMBER(10),  
  TRANSACTION_TIMESTAMP TIMESTAMP(3) WITH TIME ZONE,  
  STATUS VARCHAR2(12));
```

Table created.

```
DESC TRANSACTION_TIME_ZONE
```

Name	Null?	Type
TRANSACTION_ID		NUMBER(10)
TRANSACTION_TIMESTAMP		TIMESTAMP(3) WITH TIME ZONE
STATUS		VARCHAR2(12)

Чтобы вставить данные в столбец `TRANSACTION_TIMESTAMP`, можно использовать литерал `TIMESTAMP` со смещением часового пояса, который имеет такой вид:

```
TIMESTAMP 'YYYY-MM-DD HH:MI:SS. xxxxxxxxx {+|-} HH:MI'
```

Приведем пример вставки данных в таблицу, содержащую столбец типа `TIMESTAMP WITH TIME ZONE`:

```
INSERT INTO TRANSACTION_TIME_ZONE  
VALUES (1002, TIMESTAMP '1998-12-31 08:23:46.368 -10:30', 'NEW');
```

1 row created.

```
SELECT * FROM TRANSACTION_TIME_ZONE;
```

TRANSACTION_ID	TRANSACTION_TIMESTAMP	STATUS
1002	31-DEC-98 08.23.46.368 AM -10:30	NEW

Обратите внимание на то, что хотя тип данных называется `TIMESTAMP WITH TIME ZONE`, литерал все равно использует ключевое слово `TIMESTAMP`. Отметьте также, что смещение даты/времени в литерале указано с помощью нотации `{+|-}час:минута`.



При указании смещения часового пояса в литерале `TIMESTAMP` необходимо указать знак смещения (+ или -). Разрешенный диапазон часов для смещения часового пояса: от -12 до +13, а диапазон минут – от 0 до 59. Если ввести значение, расположенное вне этих диапазонов, будет получено сообщение об ошибке.

Если смещение часового пояса не указано, это не означает, что оно считается нулевым. В качестве часового пояса будет использоваться локальный часовой пояс, и значение смещения по умолчанию равно смещению локального часового пояса. В следующем примере для входной даты не указан часовой пояс, Oracle считает, что речь идет о локальном часовом поясе и сохраняет в столбце базы данных дату вместе с локальным часовым поясом.

```
INSERT INTO TRANSACTION_TIME_ZONE
VALUES (1003, TIMESTAMP '1999-12-31 08:23:46.368', 'NEW');

1 row created.

SELECT * FROM TRANSACTION_TIME_ZONE;

TRANSACTION_ID TRANSACTION_TIMESTAMP                                STATUS
-----
1003 31-DEC-99 08.23.46.368 AM -05:00                                NEW
```

## TIMESTAMP WITH LOCAL TIME ZONE

Тип данных `TIMESTAMP WITH LOCAL TIME ZONE` – это разновидность типа `TIMESTAMP WITH TIME ZONE`. Тип данных `TIMESTAMP WITH LOCAL TIME ZONE` задается так:

```
TIMESTAMP [ (точность долей секунды) ] WITH LOCAL TIME ZONE
```

Точность долей секунды указывается аналогично типу данных `TIMESTAMP`. Отличия `TIMESTAMP WITH LOCAL TIME ZONE` от `TIMESTAMP WITH TIME ZONE` заключаются в следующем:

- Смещение часового пояса не хранится как часть столбца данных.
- Данные, хранимые в базе данных, приводятся к часовому поясу базы данных. Для приведения введенной даты к часовому поясу базы данных входное время преобразуется во время часового пояса базы данных.
- При извлечении данных Oracle возвращает дату и время для часового пояса текущего сеанса пользователя.

Создадим таблицу со столбцом типа `TIMESTAMP`:

```
CREATE TABLE TRANSACTION_LOCAL_TIME_ZONE (
TRANSACTION_ID NUMBER(10),
TRANSACTION_TIMESTAMP(3) WITH LOCAL TIME ZONE,
STATUS VARCHAR2(12));
```

Table created.

```
DESC TRANSACTION_LOCAL_TIME_ZONE
```

Name	Null?	Type
TRANSACTION_ID		NUMBER(10)
TRANSACTION_TIMESTAMP		TIMESTAMP(3) WITH LOCAL TIME ZONE
STATUS		VARCHAR2(12)

Для типа данных `TIMESTAMP WITH LOCAL TIME ZONE` не существует специального литерала. Для ввода данных в такой столбец используем литерал `TIMESTAMP`, например:

```
INSERT INTO TRANSACTION_LOCAL_TIME_ZONE VALUES (
2001, TIMESTAMP '1998-12-31 10:00:00 -3:00', 'NEW');
```

```
1 row created.
```

```
SELECT * FROM TRANSACTION_LOCAL_TIME_ZONE;
```

TRANSACTION_ID	TRANSACTION_TIMESTAMP	STATUS
2001	31-DEC-98 08.00.00 AM	NEW

Как видите, смещение часового пояса не хранится в базе данных. Данные хранятся преобразованными во время часового пояса базы данных. То есть прежде чем введенное время сохраняется в базе данных, оно преобразуется во время часового пояса базы данных. Часовой пояс базы данных – это `-5:00`. Поэтому часовой пояс `-3:00` на два часа впереди часового пояса базы данных, и `10:00:00` в часовом поясе `-3:00` – это `08:00:00` в часовом поясе `-5:00`. Так как время приводится ко времени часового пояса базы данных, нет необходимости в хранении смещения.

## Интервалы дат и времени

Интервалы дат и времени являются неотъемлемой частью нашей повседневной жизни. К интервальному данным относится возраст человека, срок погашения облигаций или депозитного сертификата, а также срок гарантии вашей машины. До выхода `Oracle9i` все мы использовали для представления таких данных тип `NUMBER`, а логика, необходимая для обработки интервалов, должна была быть реализована на уровне приложения. `Oracle9i` вводит два новых типа данных для работы с интервалами:

- `INTERVAL YEAR TO MONTH`
- `INTERVAL DAY TO SECOND`

Далее мы поговорим об использовании этих типов данных.

### INTERVAL YEAR TO MONTH

Тип `INTERVAL YEAR TO MONTH` хранит период времени, выраженный в количестве годов и месяцев, и задается следующим образом:

```
INTERVAL YEAR [ (точность года) ] TO MONTH
```

Точность указывает количество разрядов для года; разрешенными являются значения от 0 до 9, значение по умолчанию – 2. Точность по умолчанию позволяет хранить максимальный интервал в 99 лет 11 месяцев.

Создадим таблицу с типом данных INTERVAL YEAR TO MONTH:

```
CREATE TABLE EVENT_HISTORY (  
  EVENT_ID NUMBER(10),  
  EVENT_DURATION INTERVAL YEAR TO MONTH);
```

Table created.

DESC EVENT\_HISTORY

Name	Null?	Type
EVENT_ID		NUMBER(10)
EVENT_DURATION		INTERVAL YEAR(2) TO MONTH

Для вставки данных в столбец INTERVAL YEAR TO MONTH используем функцию NUMTOYMINTERVAL (NUM-TO-YM-INTERVAL), которая преобразует значение типа NUMBER в значение типа INTERVAL YEAR TO MONTH (о ней будет подробно рассказано в разделе «Работа с временными метками и интервалами»).

```
INSERT INTO EVENT_HISTORY VALUES (5001, NUMTOYMINTERVAL(2, 'YEAR'));
```

1 row created.

```
INSERT INTO EVENT_HISTORY VALUES (5002, NUMTOYMINTERVAL(2.5, 'MONTH'));
```

1 row created.

```
SELECT * FROM EVENT_HISTORY;
```

EVENT_ID	EVENT_DURATION
5001	+02-00
5002	+00-02

Второй аргумент функции NUMTOYMINTERVAL задает единицы измерения первого аргумента. Поэтому в первом примере число 2 трактуется как 2 года, а во втором примере число 2,5 интерпретируется как 2 месяца. Обратите внимание, что дробная часть месяца игнорируется. Значение типа INTERVAL YEAR TO MONTH содержит только годы и месяцы, но не доли месяца, дробная часть месяца отбрасывается.

## INTERVAL DAY TO SECOND

Тип INTERVAL DAY TO SECOND хранит период времени, выраженный в количестве дней, часов, минут, секунд и долей секунд. Тип INTERVAL DAY TO SECOND задается так:

```
INTERVAL DAY [(точность дня)]  
TO SECOND [(точность долей секунды)]
```



Точность дня указывает количество разрядов для дня; разрешенными являются значения от 0 до 9, значение по умолчанию – 2. Точность долей секунды – это количество разрядов дробной части секунд; разрешенными являются значения от 0 до 9, значение по умолчанию – 6.

Создадим таблицу с типом данных INTERVAL DAY TO SECOND:

```
CREATE TABLE BATCH_JOB_HISTORY (  
  JOB_ID NUMBER(6),  
  JOB_DURATION INTERVAL DAY(3) TO SECOND(6));
```

Table created.

DESC BATCH\_JOB\_HISTORY

Name	Null?	Type
JOB_ID		NUMBER(6)
JOB_DURATION		INTERVAL DAY(3) TO SECOND(6)

Вставим данные в таблицу, содержащую столбец типа INTERVAL DAY TO SECOND:

```
INSERT INTO BATCH_JOB_HISTORY VALUES  
(6001, NUMTODSINTERVAL(5369.2589, 'SECOND'));
```

1 row created.

```
SELECT * FROM BATCH_JOB_HISTORY;
```

JOB_ID	JOB_DURATION
6001	+00 01:29:29.258900

Для вставки данных в столбец INTERVAL DAY TO SECOND используем функцию NUMTODSINTERVAL (NUM-TO-DS-INTERVAL), которая преобразует значение типа NUMBER в значение типа INTERVAL DAY TO SECOND (о ней будет подробно рассказано в разделе «Работа с временными метками и интервалами»).

## Литералы INTERVAL

Как вы уже знаете, Oracle поддерживает литералы DATE и TIMESTAMP. Кроме того, поддерживаются и литералы INTERVAL. Существует два интервальных типа данных и два типа соответствующих интервальных литералов: INTERVAL YEAR TO MONTH и INTERVAL DAY TO SECOND.

## Интервальные литералы YEAR TO MONTH

Интервальный литерал YEAR TO MONTH представляет период времени в терминах годов и месяцев. Литерал INTERVAL YEAR TO MONTH имеет такой вид:

```
INTERVAL 'y [- m]' YEAR[(точность года)] [TO MONTH]
```

Перечислим элементы синтаксиса:

*y*

Целое значение, указывающее годы.

*m*

Необязательное целое значение, задающее месяцы. Если вы указываете количество месяцев, то должны использовать ключевые слова TO MONTH.

*точность года*

Указывает количество разрядов, которые будут использоваться для года, по умолчанию – 2. Разрешены значения от 0 до 9.

По умолчанию точность года равна 2. Если литерал представляет период времени, превышающий 99 лет, необходимо указать большее значение точности. Целое значение для месяца, как и ключевое слово MONTH, является необязательным. Если значение для месяца задается, оно должно быть в диапазоне от 0 до 11; кроме того, в этом случае необходимо использовать ключевые слова TO MONTH.

Вставим интервальный литерал YEAR TO MONTH в столбец типа INTERVAL YEAR TO MONTH:

```
INSERT INTO EVENT_HISTORY  
VALUES (6001, INTERVAL '5-2' YEAR TO MONTH);
```

```
1 row created.
```

```
SELECT * FROM EVENT_HISTORY;
```

```
EVENT_ID EVENT_DURATION
```

```
-----  
6001 +05-02
```

Используем интервальный литерал YEAR TO MONTH для задания периода времени, равного ровно четырем годам. Заметьте, что значение для месяца не задается:

```
SELECT INTERVAL '4' YEAR FROM DUAL;
```

```
INTERVAL '4' YEAR
```

```
-----  
+04-00
```

Интервальный литерал YEAR TO MONTH можно использовать и для представления только месяцев.

```
SELECT INTERVAL '3' MONTH FROM DUAL;
```

```
INTERVAL '3' MONTH
```

```
-----  
+00-03
```

```
SELECT INTERVAL '30' MONTH FROM DUAL;
```

```
INTERVAL '30' MONTH
```

```
-----  
+02-06
```

Обратите внимание, что когда интервальный литерал YEAR TO MONTH используется для указания только месяцев, есть возможность задать значение, превышающее 11. В таких ситуациях Oracle преобразует значение в соответствующее количество лет и месяцев. Это единственная ситуация, в которой месяцев может быть больше, чем 11.

## Интервальные литералы DAY TO SECOND

Интервальный литерал DAY TO SECOND представляет период времени, определяемый в терминах дней, часов, минут и секунд. Литерал INTERVAL DAY TO SECOND имеет такой вид:

```
INTERVAL 'd [h [:m[:s]]]' DAY[(точность дня)] [TO {HOUR | MINUTE | SECOND[(точность долей секунды)]}]
```

Рассмотрим элементы синтаксиса:

*d*

Целое значение, указывающее дни.

*h*

Необязательное целое значение, задающее часы.

*m*

Необязательное целое значение, задающее минуты.

*s*

Необязательное целое значение, задающее секунды.

*точность дня*

Указывает количество разрядов, которые будут использоваться для дней, по умолчанию – 2. Разрешены значения от 0 до 9.

*точность долей секунды*

Указывает количество разрядов, которые будут использоваться для долей секунды.

По умолчанию точность месяца равна 2. Если литерал представляет период времени, превышающий 99 дней, необходимо указать большее значение. В указании точности часов и минут нет необходимости. Значения часов могут быть от 0 до 23, а значения минут – от 0 до 59. Если вы указываете доли секунды, необходимо задать для них точность. Точность долей секунды может быть от 1 до 9, а значения секунд – от 0 до 59,999999999.

Вставим интервальный литерал DAY TO SECOND в столбец типа INTERVAL DAY TO SECOND. Литерал представляет период времени в 0 дней 3 часа 16 минут и 23,45 секунды.

```
INSERT INTO BATCH_JOB_HISTORY  
VALUES (2001, INTERVAL '0 3:16:23.45' DAY TO SECOND);
```

```
1 row created.
```

```
SELECT * FROM BATCH_JOB_HISTORY;
```



```
-----
2001 +00 03:16:23.450000
-----
```

В предыдущем примере использовались все элементы интервального литерала DAY TO SECOND. Однако вы можете использовать и меньшее количество элементов. Представим несколько вполне законных вариантов:

```
SELECT INTERVAL '400' DAY(3) FROM DUAL;
INTERVAL '400' DAY(3)
-----
+400 00:00:00

SELECT INTERVAL '11:23' HOUR TO MINUTE FROM DUAL;
INTERVAL '11:23' HOURTOMINUTE
-----
+00 11:23:00

SELECT INTERVAL '11:23' MINUTE TO SECOND FROM DUAL;
INTERVAL '11:23' MINUTETOSECOND
-----
+00 00:11:23.000000

SELECT INTERVAL '20' MINUTE FROM DUAL;
INTERVAL '20' MINUTE
-----
+00 00:20:00
```

Единственное требование, которое необходимо учитывать, – при указании интервала нельзя пропускать значения в середине диапазона. Нельзя, например, указать интервал только в часах и секундах, необходимо указать и значение минут. Интервал в 4 часа 36 секунд должен быть записан как 4 часа 0 минут 36 секунд.

## Работа с временными метками и интервалами

Oracle9i предлагает ряд новых встроенных функций SQL для работы со значениями новых типов данных даты и времени и интервальных типов (табл. 6.4).

*Таблица 6.4. Новые функции Oracle9i для временных меток и интервалов*

Функция	Описание	Возвращаемый тип данных
DBTIMEZONE	Возвращает часовой пояс базы данных.	Символьный
SESSIONTIMEZONE	Возвращает часовой пояс текущего сеанса.	Символьный
SYSTIMESTAMP	Возвращает текущую дату и время в часовом поясе сеанса.	TIMESTAMP WITH TIME ZONE

Таблица 6.4 (продолжение)

Функция	Описание	Возвращаемый тип данных
CURRENT_DATE	Возвращает текущую дату в часовом поясе сеанса.	DATE
CURRENT_TIMESTAMP	Возвращает текущую дату и временную метку в часовом поясе сеанса.	TIMESTAMP WITH TIME ZONE
LOCALTIMESTAMP	Возвращает текущую дату и время в часовом поясе сеанса.	TIMESTAMP
TO_TIMESTAMP	Преобразует символьную строку в значение типа <code>TIMESTAMP</code> .	TIMESTAMP
TO_TIMESTAMP_TZ	Преобразует символьную строку в значение типа <code>TIMESTAMP WITH TIME ZONE</code> .	TIMESTAMP WITH TIME ZONE
FROM_TZ	Преобразует значение типа <code>TIMESTAMP</code> в тип <code>TIMESTAMP WITH TIME ZONE</code> .	TIMESTAMP WITH TIME ZONE
NUMTOYMINTERVAL	Преобразует число в значение типа <code>INTERVAL YEAR TO MONTH</code> .	INTERVAL YEAR TO MONTH
NUMTODSINTERVAL	Преобразует число в значение типа <code>INTERVAL DAY TO SECOND</code> .	INTERVAL DAY TO SECOND
TO_YMINTERVAL	Преобразует символьную строку в значение типа <code>INTERVAL YEAR TO MONTH</code> .	INTERVAL YEAR TO MONTH
TO_DSINTERVAL	Преобразует символьную строку в значение типа <code>INTERVAL DAY TO SECOND</code> .	INTERVAL DAY TO SECOND
TZ_OFFSET	Возвращает смещение часового пояса по отношению к UTC.	Символьный

Часовой пояс возвращается как смещение по отношению к UTC и отображается со знаком «+» или «-» и значением в формате «часы:минуты». В следующих разделах будут рассмотрены все перечисленные функции и примеры их использования.

## DBTIMEZONE

Функция `DBTIMEZONE` возвращает значение часового пояса базы данных. Эту функцию можно использовать подобно `SYSDATE`:

```
SELECT DBTIMEZONE FROM DUAL;

DBTIME
-----
-07:00
```

## SESSIONTIMEZONE

Функция `SESSIONTIMEZONE` возвращает значение часового пояса сеанса. Эту функцию также можно использовать подобно `SYSDATE`:

```
SELECT SESSIONTIMEZONE FROM DUAL;  
  
SESSIONTIMEZONE  
-----  
-06:00
```

## SYSTIMESTAMP

Функция `SYSTIMESTAMP` возвращает значение системной даты и времени, включая доли секунды и часовой пояс. Она подобна функции `SYSDATE`, только возвращаемое значение содержит дополнительную информацию о долях секунды и часовом поясе.

```
SELECT SYSTIMESTAMP FROM DUAL;  
  
SYSTIMESTAMP  
-----  
11-NOV-01 01.00.10.040438 AM -05:00
```

Функция `SYSTIMESTAMP` возвращает значение типа `TIMESTAMP WITH TIMEZONE`, и точность долей секунды всегда равна 6.

## CURRENT\_DATE

Функция `CURRENT_DATE` возвращает текущую дату и время для часового пояса сеанса. Отличие `SYSDATE` от `CURRENT_DATE` в том, что функция `SYSDATE` базируется на `DBTIMEZONE`, в то время как `CURRENT_DATE` использует `SESSIONTIMEZONE`.

```
SELECT SYSDATE, CURRENT_DATE FROM DUAL;  
  
SYSDATE                CURRENT_DATE  
-----  
11-NOV-2001 01:15:40 AM 11-NOV-2001 12:15:41 AM
```

Заметьте, что в нашем примере `CURRENT_DATE` на час отстает от `SYSDATE`. Это объясняется тем, что часовой пояс сеанса на один час отстает от часового пояса базы данных.

## CURRENT\_TIMESTAMP

Функция `CURRENT_TIMESTAMP` возвращает текущую дату, время, доли секунды и смещение часового пояса. Возвращаемое значение относится к часовому поясу сеанса. Обратите внимание на различие между функциями `SYSTIMESTAMP` и `CURRENT_TIMESTAMP` — `SYSTIMESTAMP` базируется на `DBTIMEZONE`, а `CURRENT_TIMESTAMP` — на `SESSIONTIMEZONE`.

Прототип функции `CURRENT_TIMESTAMP` выглядит так:

```
CURRENT_TIMESTAMP [(точность)]
```



Аргумент «точность» указывает точность долей секунды и является необязательным. Точность по умолчанию равна 6. Возвращаемое значение имеет тип `TIMESTAMP WITH TIME ZONE`.

```
SELECT CURRENT_TIMESTAMP FROM DUAL;
```

```
CURRENT_TIMESTAMP
```

```
-----  
11-NOV-01 01.42.40.099518 PM -06:00
```

## LOCALTIMESTAMP

Функция `LOCALTIMESTAMP` возвращает текущую дату, время и доли секунды для часового пояса сеанса. Прототип функции `LOCALTIMESTAMP` имеет вид:

```
LOCALTIMESTAMP [{точность}]
```

Аргумент «точность» указывает точность долей секунды и является необязательным. Точность по умолчанию равна 6. Возвращаемое значение имеет тип `TIMESTAMP`.

```
SELECT LOCALTIMESTAMP FROM DUAL;
```

```
LOCALTIMESTAMP
```

```
-----  
11-NOV-01 01.42.55.852724 PM
```

Единственным отличием между функциями `LOCALTIMESTAMP` и `CURRENT_TIMESTAMP` является тип возвращаемого значения. `LOCALTIMESTAMP` возвращает значение типа `TIMESTAMP`, а `CURRENT_TIMESTAMP` – значение типа `TIMESTAMP WITH TIME ZONE`.

## TO\_TIMESTAMP

Функция `TO_TIMESTAMP` похожа на функцию `TO_DATE`. Она преобразует символьную строку в значение типа `TIMESTAMP`. На вход функции `TO_TIMESTAMP` могут подаваться литералы, переменные PL/SQL или столбцы базы данных типа `CHAR` или `VARCHAR2`.



Для генерации значения типа `TIMESTAMP` также может использоваться ключевое слово `TIMESTAMP`, но ключевое слово можно применять только с литеральным значением. Функция `TO_TIMESTAMP` может работать с переменными PL/SQL и столбцами базы данных.

Прототип функции `TO_TIMESTAMP` таков:

```
TO_TIMESTAMP (строка [.формат])
```

Рассмотрим элементы синтаксиса:

### *строка*

Символьная строка или числовое значение, которые могут быть преобразованы в тип `TIMESTAMP`. Строка или числовое значение может быть литералом, значением переменной `PL/SQL` или значением столбца базы данных.

### *формат*

Указывает формат входной строки.

Формат является необязательным; если он не указан, строка интерпретируется как имеющая формат временной метки по умолчанию. Формат временной метки по умолчанию – это формат даты по умолчанию плюс время в формате `HH.MI.SS.xxxxxxxxxx`, где `xxxxxxxxxx` соответствует долям секунды. Следующий пример преобразует строку в формате временной метки по умолчанию в значение типа `TIMESTAMP`:

```
SELECT TO_TIMESTAMP('11-NOV-01 10.32.22.765488123') FROM DUAL;
```

```
TO_TIMESTAMP('11-NOV-0110.32.22.765488123')
```

```
-----  
11-NOV-01 10.32.22.765488123 AM
```

Теперь укажем в качестве второго входного параметра функции `TO_TIMESTAMP` формат:

```
SELECT TO_TIMESTAMP('12/10/01', 'MM/DD/YY') FROM DUAL;
```

```
TO_TIMESTAMP('12/10/01', 'MM/DD/YY')
```

```
-----  
10-DEC-01 12.00.00 AM
```

Заметьте, что, поскольку входная строка не содержала информации о времени, было принято время начала дня, то есть `12:00:00 A.M.`

## **TO\_TIMESTAMP\_TZ**

Функция `TO_TIMESTAMP_TZ` похожа на функцию `TO_TIMESTAMP`. Единственным отличием является тип возвращаемого значения. `TO_TIMESTAMP_TZ` возвращает значение типа `TIMESTAMP WITH TIME ZONE`. На вход функции `TO_TIMESTAMP_TZ` могут подаваться литералы, переменные `PL/SQL` или столбцы базы данных типа `CHAR` или `VARCHAR2`.

Прототип функции `TO_TIMESTAMP_TZ` таков:

```
TO_TIMESTAMP_TZ (строка [, формат])
```

Рассмотрим элементы синтаксиса:

### *строка*

Символьная строка или числовое значение, которые могут быть преобразованы в значение типа `TIMESTAMP WITH TIME ZONE`. Строка или

числовое значение может быть литералом, переменной PL/SQL или значением столбца базы данных.

### *формат*

Указывает формат входной строки.

Формат является необязательным; если он не указан, строка интерпретируется как имеющая формат по умолчанию типа `TIMESTAMP WITH TIME ZONE`. Следующий пример преобразует строку в формате по умолчанию в значение типа `TIMESTAMP WITH TIME ZONE`:

```
SELECT TO_TIMESTAMP_TZ('11-NOV-01 10.32.22.765488123 AM -06:00') FROM DUAL;  
TO_TIMESTAMP_TZ('11-NOV-0110.32.22.765488123')  
-----  
11-NOV-01 10.32.22.765488123 AM -06:00
```

Теперь укажем в качестве второго входного параметра функции `TO_TIMESTAMP_TZ` формат:

```
SELECT TO_TIMESTAMP_TZ('12/10/01', 'MM/DD/YY') FROM DUAL;  
TO_TIMESTAMP_TZ('12/10/01', 'MM/DD/YY')  
-----  
10-DEC-01 12.00.00.000000000 AM -06:00
```

Заметьте, что, поскольку входная строка не содержала информации о времени, было принято время начала дня, то есть 12:00:00 А.М.

Функция `TO_TIMESTAMP_TZ` не преобразует входную строку в формат типа данных `TIMESTAMP WITH LOCAL TIME ZONE`. Oracle не предоставляет для этих целей никакой функции. Чтобы преобразовать значение в тип `TIMESTAMP WITH LOCAL TIME ZONE`, необходимо использовать функцию `CAST`, например:

```
SELECT CAST('10-DEC-01' AS TIMESTAMP WITH LOCAL TIME ZONE) FROM DUAL;  
CAST('10-DEC-01' AS TIMESTAMP WITH LOCAL TIME ZONE)  
-----  
10-DEC-01 12.00.00 AM  
  
SELECT CAST(TO_TIMESTAMP_TZ('12/10/01', 'MM/DD/YY')  
           AS TIMESTAMP WITH LOCAL TIME ZONE)  
FROM DUAL;  
  
CAST(TO_TIMESTAMP_TZ('12/10/01', 'MM/DD/YY') AS TIMESTAMP WITH LOCAL TIME ZONE)  
-----  
10-DEC-01 12.00.00 AM
```

В первом примере входная строка имеет формат даты по умолчанию, поэтому нет необходимости в преобразовании формата. Однако во втором примере входная строка имеет формат, отличный от формата по умолчанию, поэтому необходимо применять функцию преобразования с указанием формата, чтобы строка была преобразована в значение



(например, `TIMESTAMP WITH TIME ZONE`), которое затем можно было бы привести к типу `TIMESTAMP WITH LOCAL TIME ZONE`. В зависимости от конкретных входных данных можно использовать функции `TO_DATE`, `TO_TIMESTAMP` или `TO_TIMESTAMP_TZ`.



Используемую в этих примерах функцию `CAST` не совсем корректно называть функцией SQL. Фактически `CAST` — это выражение SQL, подобное `DECODE` и `CASE`. Выражение `CAST` преобразует значение одного типа данных в значение другого типа. В первом примере выражение `CAST` преобразует символьный литерал `CHAR` в значение типа `TIMESTAMP WITH LOCAL TIME ZONE`. Во втором примере выражение `CAST` преобразует значение типа `TIMESTAMP WITH TIME ZONE` в значение `TIMESTAMP WITH LOCAL TIME ZONE`.

## FROM\_TZ

Функция `FROM_TZ` принимает на входе значения типа `TIMESTAMP` и часового пояса и преобразует их в значение типа `TIMESTAMP WITH TIME ZONE`. Прототип функции `FROM_TZ` таков:

```
FROM_TZ (значение типа TIMESTAMP, часовой пояс)
```

Рассмотрим элементы синтаксиса:

### *Значение типа TIMESTAMP*

Строковый литерал, переменная PL/SQL или столбец базы данных. Функции обязательно должно передаваться значение типа `TIMESTAMP`.

### *часовой пояс*

Строка, содержащая часовой пояс в формате `[+|-]hh:mi`.

Проиллюстрируем преобразование значения типа `TIMESTAMP` и часового пояса в значение типа `TIMESTAMP WITH TIME ZONE`:

```
SELECT FROM_TZ(TIMESTAMP '2001-12-10 08:30:00', '-5:00') FROM DUAL;  
FROM_TZ(TIMESTAMP'2001-12-1008:30:00', '-5:00')  
-----  
10-DEC-01 08.30.00.000000000 AM -05:00
```

## NUMTOYMINTERVAL

Функция `NUMTOYMINTERVAL` (`NUM-TO-YM-INTERVAL`) преобразует число в литерал `INTERVAL YEAR TO MONTH`. Прототип функции `NUMTOYMINTERVAL` таков:

```
NUMTOYMINTERVAL (число, единицы)
```

Рассмотрим элементы синтаксиса:

### *число*

Числовой литерал или выражение, преобразуемое в число.

### *единицы*

Символьная строка, указывающая единицы измерения для первого параметра, может принимать значения 'YEAR' или 'MONTH'. Регистр не имеет значения.

В следующем примере строка вставляется в таблицу, содержащую столбец типа INTERVAL YEAR TO MONTH. Функция NUMTOYMINTERVAL применяется для преобразования числа в значение типа INTERVAL YEAR TO MONTH.

```
INSERT INTO EVENT_HISTORY VALUES (5001, NUMTOYMINTERVAL(2, 'YEAR'));
```

## **NUMTODSINTERVAL**

Функция NUMTODSINTERVAL (NUM-TO-DS-INTERVAL) преобразует число в литерал INTERVAL DAY TO SECOND. Заголовок функции NUMTODSINTERVAL таков:

```
NUMTODSINTERVAL (число, единицы)
```

Рассмотрим элементы синтаксиса:

### *число*

Числовой литерал или выражение, преобразуемое в число.

### *единицы*

Символьная строка, указывающая единицы измерения для первого параметра, может принимать значения 'DAY', 'HOUR', 'MINUTE' или 'SECOND'. Регистр не имеет значения.

В следующем примере строка вставляется в таблицу, содержащую столбец типа INTERVAL DAY TO SECOND. Функция NUMTODSINTERVAL применяется для преобразования числа в значение типа INTERVAL DAY TO SECOND.

```
INSERT INTO BATCH_JOB_HISTORY VALUES  
(6001, NUMTODSINTERVAL(5369.2589, 'SECOND'));
```

## **TO\_YMINTERVAL**

Функция TO\_YMINTERVAL очень похожа на функцию TO\_DATE. Она преобразует символьную строку в тип INTERVAL YEAR TO MONTH. На вход функции может подаваться литерал, переменная PL/SQL или столбец базы данных типа CHAR или VARCHAR2.

Прототип функции TO\_YMINTERVAL таков:

```
TO_YMINTERVAL (строка)
```

Рассмотрим элемент синтаксиса:

### *строка*

Строковый литерал, переменная PL/SQL или столбец базы данных. Входная строка должна содержать числовые или символьные данные, преобразуемые в значение типа INTERVAL YEAR TO MONTH, и соответствовать формату Y-M, то есть значения года и месяца должны



быть разделены тире (-). Строка должна обязательно содержать все компоненты (год, месяц и тире).

Вставим строку в таблицу, содержащую столбец типа INTERVAL YEAR TO MONTH. Функция TO\_YMINTERVAL применяется для преобразования строки в значение типа INTERVAL YEAR TO MONTH.

```
INSERT INTO EVENT_HISTORY VALUES (5001, TO_YMINTERVAL('02-04'));
```

В данном случае строка '02-04' представляет интервал в 2 года и 4 месяца.

## TO\_DSINTERVAL

Функция TO\_DSINTERVAL также похожа на функцию TO\_DATE. Она преобразует символьную строку в значение типа INTERVAL DAY TO SECOND. На вход функции может подаваться литерал, переменная PL/SQL или столбец базы данных типа CHAR или VARCHAR2.

Прототип функции TO\_DSINTERVAL таков:

```
TO_DSINTERVAL (строка)
```

Рассмотрим элемент синтаксиса:

*строка*

Строковый литерал, переменная PL/SQL или столбец базы данных, содержащие числовые или символьные данные, преобразуемые в значение типа INTERVAL DAY TO SECOND. Строка, подаваемая на вход функции, должна быть в формате D HH:MI:SS. Значение дня отделено пробелом от значения времени, выраженного в часах, минутах и секундах, разделенных двоеточиями (:). Для выполнения преобразования в значение INTERVAL DAY TO SECOND строка должна обязательно содержать все компоненты.

Вставим строку в таблицу, содержащую столбец типа INTERVAL DAY TO SECOND. Функция TO\_DSINTERVAL применяется для преобразования строки в значение типа INTERVAL DAY TO SECOND.

```
INSERT INTO BATCH_JOB_HISTORY VALUES (6001, TO_DSINTERVAL('0 2:30:43'));
```

В данном примере строка '0 2:30:43' представляет собой интервал в 0 дней 2 часа 30 минут и 43 секунды.

## TZ\_OFFSET

Функция TZ\_OFFSET возвращает смещение часового пояса для передаваемой в качестве входного параметра даты. Прототип функции TZ\_OFFSET таков:

```
TZ_OFFSET (название_часового_пояса | смещение_часового_пояса | DBTIMEZONE |  
SESSIONTIMEZONE)
```



Рассмотрим элементы синтаксиса:

*название часового пояса*

Строка, содержащая название часового пояса. Все часовые пояса имеют свое название. Чтобы получить список действительных названий часовых поясов, можно выполнить запрос к динамическому представлению V\$TIMEZONE\_NAMES.

*смещение часового пояса*

Строка, содержащая смещение часового пояса в виде {+|-} hh:mi, то есть часы и минуты, перед которыми стоит знак «+» или «-».

**DBTIMEZONE**

DBTIMEZONE – это встроенная функция, возвращающая часовой пояс базы данных.

**SESSIONTIMEZONE**

SESSIONTIMEZONE – это встроенная функция, возвращающая часовой пояс сеанса.

Приведем пример использования функции TZ\_OFFSET:

```
SELECT TZ_OFFSET('US/Pacific'), TZ_OFFSET('EST'), TZ_OFFSET('+6:30') FROM  
DUAL;
```

```
TZ_OFFSET TZ_OFFSET TZ_OFFSET
```

```
-----
```

```
-08:00 -05:00 +06:30
```

Обратите внимание, что можно использовать как полные названия часовых поясов, такие как «US/Eastern» и «US/Pacific», так и стандартные сокращения: «EST», «PST» и др. В заключение приведем пример применения функций DBTIMEZONE и SESSIONTIMEZONE в качестве параметра функции TZ\_OFFSET:

```
SELECT TZ_OFFSET(DBTIMEZONE), TZ_OFFSET(SESSIONTIMEZONE) FROM DUAL;
```

```
TZ_OFFSET TZ_OFFSET
```

```
-----
```

```
-07:00 -06:00
```

## Операции над множествами

Бывают случаи, когда необходимо скомбинировать результаты двух или более операторов SELECT. SQL позволяет разрешить такие ситуации с помощью операций над множествами. Результат каждого оператора SELECT можно интерпретировать как множество и применять к таким множествам соответствующие операции SQL для достижения нужного результата. Oracle SQL поддерживает четыре операции над множествами:

- UNION ALL
- UNION
- MINUS
- INTERSECT

Операторы SQL, содержащие операции над множествами, называются *составными запросами (compound queries)*, а каждый оператор SELECT составного запроса называется *компонентом*. Два оператора SELECT могут образовать составной запрос при помощи операции над множествами, только если они удовлетворяют следующим двум условиям:

1. Результирующие множества двух запросов должны иметь одинаковое количество столбцов.
2. Тип данных каждого столбца второго результирующего множества должен совпадать с типом данных соответствующего столбца первого результирующего множества.



Типы данных не обязаны совпадать, если Oracle может автоматически преобразовать типы данных второго результирующего множества (используя неявное приведение типов) к типам, совместимым с типами данных первого результирующего множества.

Эти условия называют условиями *совместимости по операции слияния* (*union compatibility conditions*). Термин «совместимость по операции слияния» используется, даже если речь идет о другой операции над множествами. Операции над множествами также часто называют *вертикальными объединениями*, так как результат объединяет данные нескольких операторов SELECT по столбцам, а не по строкам. Конструкция запроса, содержащего операцию над множествами, выглядит следующим образом:

```
<запрос-компонент>  
{UNION | UNION ALL | MINUS | INTERSECT}  
<запрос-компонент>
```

Ключевые слова UNION, UNION ALL, MINUS и INTERSECT — это операторы работы с множествами. В составном запросе может быть и более двух компонентов, при этом количество операторов работы с множествами всегда на единицу меньше количества запросов-компонентов.

В следующих разделах мы поговорим о синтаксисе, примерах использования, правилах и ограничениях, налагаемых на четыре операции над множествами.

## Операторы работы с множествами

Приведем краткое описание четырех операций над множествами, поддерживаемых Oracle SQL:

### UNION ALL

Объединяет результаты двух операторов SELECT в одно результирующее множество.

### UNION

Объединяет результаты двух операторов SELECT в одно результирующее множество и удаляет из него все дубликаты.

### MINUS

Берет результирующее множество одного оператора SELECT и удаляет из него строки, возвращенные вторым оператором SELECT.

### INTERSECT

Возвращает только те строки, которые возвращены каждым из операторов SELECT.

Прежде чем переходить к подробному рассказу об этих операторах, давайте рассмотрим два запроса, которые будут использованы как запросы-компоненты в последующих примерах. Первый запрос извлекает всех клиентов региона с номером 5:

```
SELECT CUST_NBR, NAME  
FROM CUSTOMER
```



```
WHERE REGION_ID = 5;
```

```
CUST_NBR NAME
```

```
-----  
1 Cooper Industries  
2 Emblazon Corp.  
3 Ditech Corp.  
4 Flowtech Inc.  
5 Gentech Industries
```

Второй запрос извлекает всех клиентов, для которых MARTIN является торговым представителем:

```
SELECT C.CUST_NBR, C.NAME  
FROM CUSTOMER C  
WHERE C.CUST_NBR IN (SELECT O.CUST_NBR  
                     FROM CUST_ORDER O, EMPLOYEE E  
                     WHERE O.SALES_EMP_ID = E.EMP_ID  
                     AND E.LNAME = 'MARTIN');
```

```
CUST_NBR NAME
```

```
-----  
4 Flowtech Inc.  
8 Zantech Inc.
```

Если вы посмотрите на результаты, возвращенные запросами, то заметите одну общую строку (относящуюся к Flowtech Inc.). В следующих разделах будут обсуждаться эффекты различных операций над множествами, применяемых к этим результирующим множествам.

## UNION ALL

Оператор UNION ALL соединяет результирующие множества двух запросов-компонентов. Эта операция вернет строки, возвращенные любым из двух запросов. Рассмотрим пример использования операции UNION ALL:

```
SELECT CUST_NBR, NAME  
FROM CUSTOMER  
WHERE REGION_ID = 5  
UNION ALL  
SELECT C.CUST_NBR, C.NAME  
FROM CUSTOMER C  
WHERE C.CUST_NBR IN (SELECT O.CUST_NBR  
                     FROM CUST_ORDER O, EMPLOYEE E  
                     WHERE O.SALES_EMP_ID = E.EMP_ID  
                     AND E.LNAME = 'MARTIN');
```

```
CUST_NBR NAME
```

```
-----  
1 Cooper Industries  
2 Emblazon Corp.  
3 Ditech Corp.
```

4 Flowtech Inc.  
5 Gentech Industries  
4 Flowtech Inc.  
8 Zantech Inc.

7 rows selected.

Как видно из результирующего множества, есть заказчик, который извлечен каждым из операторов SELECT, поэтому он дважды присутствует в результате. Оператор UNION ALL просто объединяет выходную информацию запросов-компонентов, не заботясь о дубликатах в итоговом результирующем множестве.

## UNION

Оператор UNION возвращает все различные строки, извлеченные двумя запросами-компонентами. При объединении строк, возвращенных запросами, операция UNION удаляет дубликаты. Рассмотрим пример использования операции UNION:

```
SELECT CUST_NBR, NAME
FROM CUSTOMER
WHERE REGION_ID = 5
UNION
SELECT C.CUST_NBR, C.NAME
FROM CUSTOMER C
WHERE C.CUST_NBR IN (SELECT O.CUST_NBR
                     FROM CUST_ORDER O, EMPLOYEE E
                     WHERE O.SALES_EMP_ID = E.EMP_ID
                     AND E.LNAME = 'MARTIN');
```

CUST\_NBR NAME

-----  
1 Cooper Industries  
2 Emblazon Corp.  
3 Ditech Corp.  
4 Flowtech Inc.  
5 Gentech Industries  
8 Zantech Inc.

6 rows selected.

Этот запрос является модификацией предыдущего, где ключевое слово UNION ALL заменено на UNION. Обратите внимание, что результирующее множество содержит только различные строки (без повторений). Для удаления строк-дубликатов операция UNION должна выполнить некоторую дополнительную (по сравнению с UNION ALL) работу – результирующее множество сортируется и фильтруется. Если вы посмотрите внимательно, то заметите, что результирующее множество операции UNION ALL не отсортировано, в то время как результат выполнения операции UNION отсортирован. Выполнение дополнительных задач влияет на про-

изводительность операции UNION. Запрос с операцией UNION займет больше времени, чем такой же запрос с операцией UNION ALL, даже если в результирующем множестве нет повторений. Поэтому если особой необходимости в выводе только различных строк нет, следует использовать UNION ALL для улучшения производительности запроса.

## INTERSECT

Оператор INTERSECT возвращает только строки, возвращенные как одним, так и другим запросом-компонентом. Можно сказать, что оператор UNION, возвращающий все строки, извлеченные обоими запросами, работает как OR, а INTERSECT — как AND. Например:

```
SELECT CUST_NBR, NAME
FROM CUSTOMER
WHERE REGION_ID = 5
INTERSECT
SELECT C.CUST_NBR, C.NAME
FROM CUSTOMER C
WHERE C.CUST_NBR IN (SELECT O.CUST_NBR
                     FROM CUST_ORDER O, EMPLOYEE E
                     WHERE O.SALES_EMP_ID = E.EMP_ID
                     AND E.LNAME = 'MARTIN');
```

CUST\_NBR NAME

-----  
4 Flowtech Inc.

Как вы видели в предыдущем разделе, единственным клиентом, извлеченным и первым и вторым запросом, был «Flowtech Inc.». Поэтому оператор INTERSECT возвращает ровно одну строку.

## MINUS

Операция MINUS возвращает все строки первого оператора SELECT, которые не входят в результирующее множество второго SELECT, например:

```
SELECT CUST_NBR, NAME
FROM CUSTOMER
WHERE REGION_ID = 5
MINUS
SELECT C.CUST_NBR, C.NAME
FROM CUSTOMER C
WHERE C.CUST_NBR IN (SELECT O.CUST_NBR
                     FROM CUST_ORDER O, EMPLOYEE E
                     WHERE O.SALES_EMP_ID = E.EMP_ID
                     AND E.LNAME = 'MARTIN');
```

CUST\_NBR NAME

-----  
1 Cooper Industries



- 2 Emblazon Corp.
- 3 Ditech Corp.
- 5 Gentech Industries

Вас может удивить, что в выводе не присутствует «Zantech Inc.». Запомните важную вещь – запросы-компоненты, связанные операцией над множествами, выполняются сверху вниз. Результат операций UNION, UNION ALL и INTERSECT не изменится при изменении порядка компонентов. А вот результат выполнения оператора MINUS изменится. Если переписать предыдущий запрос, поменяв местами два запроса SELECT, то результат будет совершенно другим:

```
SELECT C.CUST_NBR, C.NAME
FROM CUSTOMER C
WHERE C.CUST_NBR IN (SELECT O.CUST_NBR
                     FROM CUST_ORDER O, EMPLOYEE E
                     WHERE O.SALES_EMP_ID = E.EMP_ID
                     AND E.LNAME = 'MARTIN')

MINUS

SELECT CUST_NBR, NAME
FROM CUSTOMER
WHERE REGION_ID = 5;

CUST_NBR NAME
```

8 Zantech Inc.

Строка для «Flowtech Inc.» возвращается обоими запросами, поэтому в первом примере с MINUS первый запрос-компонент добавляет ее в результирующее множество, а второй запрос-компонент удаляет ее оттуда. Второй пример полностью меняет операцию MINUS. Первый компонент добавляет в результирующее множество строки для «Flowtech Inc.» и «Zantech Inc.». Второй запрос-компонент указывает строки для удаления. «Flowtech Inc.» удаляется из результирующего множества, и остается единственная строка – «Zantech Inc.».



При выполнении операции MINUS второй оператор SELECT может вернуть строки, не возвращенные первым. Такие строки не включаются в результирующее множество.

## Использование операций над множествами для сравнения двух таблиц

Разработчикам и администраторам баз данных часто требуется сравнивать содержимое таблиц, чтобы определить, содержат ли они одинаковые данные. Такая необходимость особенно часто возникает в условиях тестирования, когда разработчики сравнивают набор данных, сгенерированный проверяемой программой, с заведомо правильным

набором данных. Сравнение таблиц также полезно для автоматического тестирования, когда реальные результаты сравниваются с ожидаемыми. Операции над множествами в SQL предлагают интересное решение проблемы сравнения двух таблиц.

В следующем запросе использованы сразу два оператора: MINUS и UNION ALL для проверки равенства двух таблиц. Запрос основывается на том, что каждая из таблиц имеет или первичный ключ, или хотя бы один уникальный индекс.

```
(SELECT * FROM CUSTOMER_KNOWN_GOOD
MINUS
SELECT * FROM CUSTOMER_TEST)
UNION ALL
(SELECT * FROM CUSTOMER_TEST
MINUS
SELECT * FROM CUSTOMER_KNOWN_GOOD);
```

Давайте немного поговорим о том, как работает этот запрос. Будем рассматривать его как слияние двух составных запросов. Скобки гарантируют, что обе операции MINUS будут выполнены прежде, чем операция UNION ALL. Результатом первого оператора MINUS являются те строки таблицы CUSTOMER\_KNOWN\_GOOD, которые не вошли в таблицу CUSTOMER\_TEST. Результат выполнения второго оператора MINUS — те строки таблицы CUSTOMER\_TEST, которые не входят в таблицу CUSTOMER\_KNOWN\_GOOD. Оператор UNION ALL просто объединяет эти два результата. Если запрос не возвращает ни одной строки, это означает, что таблицы полностью идентичны. Если же какие-то строки возвращаются, они представляют собой различия между таблицами CUSTOMER\_TEST и CUSTOMER\_KNOWN\_GOOD.

Если есть вероятность того, что одна из таблиц (или обе) содержит одинаковые строки, необходимо использовать более общую форму запроса для проверки равенства таблиц. Новый запрос будет выявлять дубликаты:

```
(SELECT C1.*, COUNT(*)
FROM CUSTOMER_KNOWN_GOOD
GROUP BY C1.CUST_NBR, C1.NAME...)
MINUS
(SELECT C2.*, COUNT(*)
FROM CUSTOMER_TEST C2
GROUP BY C2.CUST_NBR, C2.NAME...)
UNION ALL
(SELECT C3.*, COUNT(*)
FROM CUSTOMER_TEST C3
GROUP BY C3.CUST_NBR, C3.NAME...)
MINUS
(SELECT C4.*, COUNT(*)
FROM CUSTOMER_KNOWN_GOOD C4
GROUP BY C4.CUST_NBR, C4.NAME...)
```

Запрос стал сложным! Инструкция GROUP BY (см. главу 4) для каждого оператора SELECT должна выводить *все* столбцы запрашиваемой таблицы. Повторяющиеся строки будут сгруппированы вместе, и счетчик отразит количество дубликатов. Если количество дубликатов в обеих таблицах одинаково, операции MINUS уравнивают эти строки. Если же количества дубликатов не совпадают или есть отличающиеся строки, соответствующие строки попадут в результирующее множество запроса.

Посмотрим на работу данного запроса на примере. Будем работать с такими таблицами и данными:

```
DESC CUSTOMER_KNOWN_GOOD
```

Name	Null?	Type
CUST_NBR	NOT NULL	NUMBER(5)
NAME	NOT NULL	VARCHAR2(30)

```
SELECT * FROM CUSTOMER_KNOWN_GOOD;
```

CUST_NBR	NAME
1	Sony
1	Sony
2	Samsung
3	Panasonic
3	Panasonic
3	Panasonic

6 rows selected.

```
DESC CUSTOMER_TEST
```

Name	Null?	Type
CUST_NBR	NOT NULL	NUMBER(5)
NAME	NOT NULL	VARCHAR2(30)

```
SELECT * FROM CUSTOMER_TEST;
```

CUST_NBR	NAME
1	Sony
1	Sony
2	Samsung
2	Samsung
3	Panasonic

Как видите, таблицы CUSTOMER\_KNOWN\_GOOD и CUSTOMER\_TEST имеют одинаковую структуру, но содержат разные данные. Кроме того, ни одна из таблиц не имеет ни первичного, ни уникального ключа, и в обеих есть повторяющиеся строки. Используем приведенный ниже запрос для эффективного сравнения двух таких таблиц:

```
(SELECT C1.*, COUNT(*)
FROM CUSTOMER_KNOWN_GOOD C1
```



```

GROUP BY C1.CUST_NBR, C1.NAME
MINUS
SELECT C2.*, COUNT(*)
FROM CUSTOMER_TEST C2
GROUP BY C2.CUST_NBR, C2.NAME)
UNION ALL
(SELECT C3.*, COUNT(*)
FROM CUSTOMER_TEST C3
GROUP BY C3.CUST_NBR, C3.NAME
MINUS
SELECT C4.*, COUNT(*)
FROM CUSTOMER_KNOWN_GOOD C4
GROUP BY C4.CUST_NBR, C4.NAME);

```

CUST_NBR	NAME	COUNT(*)
2	Samsung	1
3	Panasonic	3
2	Samsung	2
3	Panasonic	1

Результаты показывают, что одна таблица (CUSTOMER\_KNOWN\_GOOD) содержит одну запись для «Samsung», а другая таблица (CUSTOMER\_TEST) содержит две записи для этого заказчика. Более того, таблица CUSTOMER\_KNOWN\_GOOD содержит три записи для «Panasonic», а CUSTOMER\_TEST — одну. Обе таблицы имеют одинаковое количество строк (2) для «Sony», поэтому «Sony» отсутствует в результирующем множестве.



В таблицах, имеющих первичный ключ или хотя бы один уникальный индекс, невозможны дубликаты. Для сравнения таких таблиц используйте краткую форму запроса.

## Использование NULL в составных запросах

В начале главы упоминались условия совместимости по операции слияния. Этот вопрос особенно интересен, когда речь заходит об использовании значений NULL. Как вы знаете, NULL не имеет типа и может использоваться вместо значения любого типа. Если умышленно выбрать NULL как значение столбца запроса-компонента, у Oracle не будет двух типов для сравнения на предмет совместимости запросов-компонентов. Для символьных столбцов это не вызывает проблем, например:

```

SELECT 1 NUM, 'DEFINITE' STRING FROM DUAL
UNION
SELECT 2 NUM, NULL STRING FROM DUAL;

```

NUM	STRING
1	DEFINITE
2	

Заметьте, что Oracle считает символьную строку 'DEFINITE' из первого запроса совместимой со значением NULL, предоставленным для соответствующего столбца вторым запросом. Но если значение NULL устанавливается для столбца типа NUMBER или DATE, необходимо явно указать Oracle поведение для значения NULL. Иначе будет выведено сообщение об ошибке, например:

```
SELECT 1 NUM, 'DEFINITE' STRING FROM DUAL
UNION
SELECT NULL NUM, 'UNKNOWN' STRING FROM DUAL;

SELECT 1 NUM, 'DEFINITE' STRING FROM DUAL
*
ERROR at line 1:
ORA-01790: expression must have same datatype as corresponding expression
```

Обратите внимание на то, что использование NULL во втором запросе-компоненте приводит к несоответствию типов данных первого столбца первого и второго запросов. Использование NULL для столбца типа DATE приводит к той же проблеме, например:

```
SELECT 1 NUM, SYSDATE DATES FROM DUAL
UNION
SELECT 2 NUM, NULL DATES FROM DUAL;
SELECT 1 NUM, SYSDATE DATES FROM DUAL
*
ERROR at line 1:
ORA-01790: expression must have same datatype as corresponding expression
```

В подобных случаях необходимо привести NULL к подходящему типу данных, например:

```
SELECT 1 NUM, 'DEFINITE' STRING FROM DUAL
UNION
SELECT TO_NUMBER(NULL) NUM, 'UNKNOWN' STRING FROM DUAL;
```

```
NUM STRING
```

```
-----
1 DEFINITE
   UNKNOWN
```

```
SELECT 1 NUM, SYSDATE DATES FROM DUAL
UNION
SELECT 2 NUM, TO_DATE(NULL) DATES FROM DUAL;
```

```
NUM DATES
```

```
-----
1 06-JAN-02
2
```

Проблемы, возникающие с совместимостью по условию слияния при использовании NULL, встречаются в Oracle8i, но в Oracle9i их уже нет:

```

SELECT 1 NUM, 'DEFINITE' STRING FROM DUAL
UNION
SELECT NULL NUM, 'UNKNOWN' STRING FROM DUAL;

```

```

NUM STRING
-----

```

```

1 DEFINITE
   UNKNOWN

```

```

SELECT 1 NUM, SYSDATE DATES FROM DUAL
UNION
SELECT 2 NUM, NULL DATES FROM DUAL;

```

```

NUM DATES
-----

```

```

1 06-JAN-02
2

```

Oracle9i достаточно умен, чтобы понять, как именно интерпретировать значение NULL в составном запросе.

## Правила и ограничения, налагаемые на операции над множествами

Наряду с условиями совместимости по операции слияния, описанными в начале главы, существует еще ряд других правил и ограничений, относящихся к операциям над множествами. В этом разделе остановимся на них подробнее.

Названия столбцов результирующего множества берутся из первого оператора SELECT:

```

SELECT CUST_NBR "Customer ID", NAME "Customer Name"
FROM CUSTOMER
WHERE REGION_ID = 5
UNION
SELECT C.CUST_NBR, C.NAME
FROM CUSTOMER C
WHERE C.CUST_NBR IN (SELECT O.CUST_NBR
                     FROM CUST_ORDER O, EMPLOYEE E
                     WHERE O.SALES_EMP_ID = E.EMP_ID
                     AND E.LNAME = 'MARTIN');

```

```

Customer ID Customer Name
-----

```

```

1 Cooper Industries
2 Emblazon Corp.
3 Ditech Corp.
4 Flowtech Inc.
5 Gentech Industries
8 Zantech Inc.

```

```

6 rows selected.

```



Хотя оба запроса используют псевдонимы столбцов, результирующее множество получает названия столбцов первого оператора SELECT. То же происходит и при создании представления на основе операции над множествами. Названия столбцов представления берутся из первого запроса-компонента:

```
CREATE VIEW V_TEST_CUST AS
SELECT CUST_NBR "Customer ID", NAME "Customer Name"
FROM CUSTOMER
WHERE REGION_ID = 5
UNION
SELECT C.CUST_NBR, C.NAME
FROM CUSTOMER C
WHERE C.CUST_NBR IN (SELECT O.CUST_NBR
                     FROM CUST_ORDER O, EMPLOYEE E
                     WHERE O.SALES_EMP_ID = E.EMP_ID
                     AND E.LNAME = 'MARTIN');
```

View created.

DESC V\_TEST\_CUST

Name	Null?	Type
Customer_ID		NUMBER
Customer_Name		VARCHAR2(45)

Если в запросе, содержащем операции над множествами, вы хотите использовать инструкцию ORDER BY, необходимо поместить ее в самый конец оператора. Инструкция ORDER BY может быть применена только один раз в конце составного запроса. Запросы-компоненты не могут содержать индивидуальных инструкций ORDER BY, например:

```
SELECT CUST_NBR, NAME
FROM CUSTOMER
WHERE REGION_ID = 5
UNION
SELECT EMP_ID, LNAME
FROM EMPLOYEE
WHERE LNAME = 'MARTIN'
ORDER BY CUST_NBR;
```

CUST_NBR	NAME
1	Cooper Industries
2	Emblazon Corp.
3	Ditech Corp.
4	Flowtech Inc.
5	Gentech Industries
7654	MARTIN

6 rows selected.

Заметьте, что название столбца, используемое инструкцией ORDER BY, взято из первого оператора SELECT. Упорядочить эти результаты по значению столбца EMP\_ID невозможно. Если попытаться выполнить инструкцию ORDER BY EMP\_ID, будет выдана ошибка, например:

```
SELECT CUST_NBR, NAME
FROM CUSTOMER
WHERE REGION_ID = 5
UNION
SELECT EMP_ID, LNAME
FROM EMPLOYEE
WHERE LNAME = 'MARTIN' ORDER BY EMP_ID;
ORDER BY EMP_ID
*
ERROR at line 8:
ORA-00904: invalid column name
```

Инструкция ORDER BY не распознает названия столбцов второго запроса-компонента. Чтобы избежать путаницы с названиями столбцов, принято указывать в инструкции ORDER BY позиции столбцов:

```
SELECT CUST_NBR, NAME
FROM CUSTOMER
WHERE REGION_ID = 5
UNION
SELECT EMP_ID, LNAME
FROM EMPLOYEE
WHERE LNAME = 'MARTIN'
ORDER BY 1;
```

```
CUST_NBR NAME
```

```
-----
1 Cooper Industries
2 Emblazon Corp.
3 Ditech Corp.
4 Flowtech Inc.
5 Gentech Industries
7654 MARTIN
```

6 rows selected.



В отличие от ORDER BY, использование инструкций GROUP BY и HAVING в запросах-компонентах разрешено.

Запросы-компоненты выполняются сверху вниз. Если вы хотите изменить последовательность выполнения, используйте скобки. Рассмотрим пример:

```
SELECT * FROM SUPPLIER_GOOD
UNION
SELECT * FROM SUPPLIER_TEST
```

```
MINUS
SELECT * FROM SUPPLIER;
```

```
SUPPLIER_ID NAME
```

```
-----
4 Toshiba
```

Oracle сначала производит слияние таблиц SUPPLIER\_GOOD и SUPPLIER\_TEST, а затем выполняет оператор MINUS для результата слияния и таблицы SUPPLIER. Если вы хотите, чтобы сначала выполнялась операция MINUS для таблиц SUPPLIER\_TEST и SUPPLIER, а затем уже производилось слияние таблицы SUPPLIER\_GOOD с результатом операции MINUS, то должны использовать скобки:

```
SELECT * FROM SUPPLIER_GOOD
UNION
(SELECT * FROM SUPPLIER_TEST
MINUS
SELECT * FROM SUPPLIER);
```

```
SUPPLIER_ID NAME
```

```
-----
1 Sony
2 Samsung
3 Panasonic
4 Toshiba
```

Скобки обеспечивают первоочередное выполнение операции MINUS (до операции UNION). Обратите внимание на разницу результатов этого и предыдущего примеров.

Далее перечислены некоторые простые правила, ограничения и замечания, не требующие дополнительных пояснений и примеров:

- Операции над множествами не разрешены для столбцов типа BLOB, CLOB, BFILE и VARRAY, а также для столбцов вложенных таблиц.
- Так как операции UNION, INTERSECT и MINUS подразумевают упорядочивание, они не разрешены для столбцов типа LONG. При этом операция UNION ALL для типа LONG разрешена.
- Операции над множествами не разрешены для операторов SELECT, содержащих выражения для коллекций TABLE.
- Операторы SELECT, к которым применяются операции над множествами, не могут использовать инструкцию FOR UPDATE.
- Количество и размер столбцов списка SELECT для запросов-компонентов ограничены размером блока базы данных. Общее количество байт в выбираемых столбцах не должно превышать размера одного блока базы данных.



# 8

## Иерархические запросы

Реляционная база данных базируется на множествах, при этом каждая таблица представляет некоторое множество данных. Но существуют такие виды информации, которые невозможно непосредственно привести к подобной структуре. Подумайте, например, о схеме структуры предприятия, спецификации материалов на заводе-изготовителе и в сборочном цехе или о генеалогическом древе. Такая информация по природе своей является иерархической, и ее удобнее представлять в древовидной структуре. В этой главе будут описаны способы представления иерархической информации в реляционной таблице. Также будут подробно рассмотрены различные конструкции SQL, которые необходимо использовать для извлечения иерархической информации из реляционной таблицы.

### Представление иерархической информации

Давайте рассмотрим на примере, как можно представить иерархическую информацию в реляционной базе данных. Для примеров будем использовать схему организации, на которой показано, как один сотрудник связан с другими в большой компании (рис. 8.1).

На рис. 8.1 представлена иерархия служащих компании. Информация о служащем, его руководителе и соответствующее отношение должны быть представлены в одной таблице EMPLOYEE, как показано на диаграмме «объект-отношение» (рис. 8.2).

На диаграмме видно, что таблица EMPLOYEE ссылается сама на себя. Столбец MANAGER\_EMP\_ID ссылается на столбец EMP\_ID той же таблицы. Для представления иерархических данных необходимо использовать

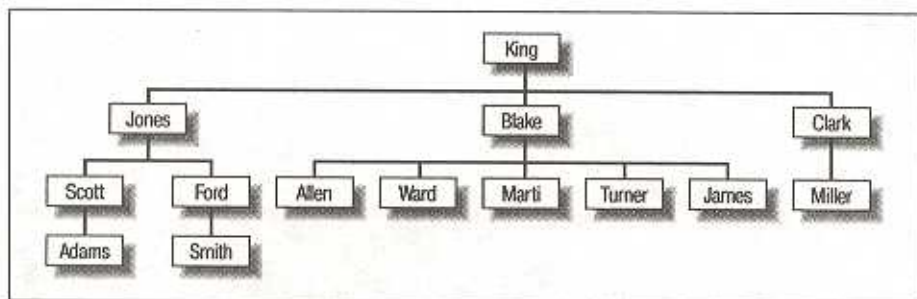


Рис. 8.1. Схема организации

такое отношение, в котором один столбец таблицы ссылается на другой столбец той же таблицы. Подобное отношение, реализованное при помощи ограничения базы данных, называется *ограничением самоссылочной целостности (self-referential integrity constraint)*. Соответствующий оператор CREATE TABLE будет выглядеть следующим образом:

```

CREATE TABLE EMPLOYEE (
EMP_ID          NUMBER (4) CONSTRAINT EMP_PK PRIMARY KEY,
FNAME           VARCHAR2 (15) NOT NULL,
LNAME           VARCHAR2 (15) NOT NULL,
DEPT_ID         NUMBER (2) NOT NULL,
MANAGER_EMP_ID  NUMBER (4) CONSTRAINT EMP_FK REFERENCES EMPLOYEE(EMP_ID),
SALARY          NUMBER (7,2) NOT NULL,
HIRE_DATE       DATE NOT NULL,
JOB_ID          NUMBER (3));
  
```

В примерах этой главы мы будем работать со следующими тестовыми данными:

```

SELECT EMP_ID, LNAME, DEPT_ID, MANAGER_EMP_ID, SALARY, HIRE_DATE
FROM EMPLOYEE;
  
```

EMP_ID	LNAME	DEPT_ID	MANAGER_EMP_ID	SALARY	HIRE_DATE
7369	SMITH	20	7902	800	17-DEC-80
7499	ALLEN	30	7698	1600	20-FEB-81

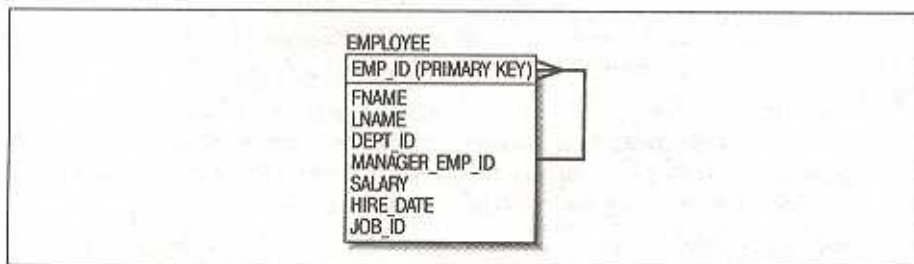


Рис. 8.2. Диаграмма «объект-отношение» для таблицы EMPLOYEE

7521	WARD	30	7698	1250	22-FEB-81
7566	JONES	20	7839	2000	02-APR-81
7654	MARTIN	30	7698	1250	28-SEP-81
7698	BLAKE	30	7839	2850	01-MAY-80
7782	CLARK	10	7839	2450	09-JUN-81
7788	SCOTT	20	7566	3000	19-APR-87
7839	KING	10		5000	17-NOV-81
7844	TURNER	30	7698	1500	08-SEP-81
7876	ADAMS	20	7788	1100	23-MAY-87
7900	JAMES	30	7698	950	03-DEC-81
7902	FORD	20	7566	3000	03-DEC-81
7934	MILLER	10	7782	1300	23-JAN-82

В таблице EMPLOYEE особого внимания заслуживают следующие особенности:

- Столбец `MANAGER_EMP_ID`
- Ограничение `EMP_FK`

Столбец `MANAGER_EMP_ID` хранит значение из столбца `EMP_ID` руководителя данного служащего. Например, `MANAGER_EMP_ID` для служащего Smith равен 7902, что означает, что руководителем Smith является Ford. Для служащего King значение `MANAGER_EMP_ID` не указано, и значит, King – это сотрудник, занимающий самое высокое положение. Чтобы иметь возможность представлять наивысшего начальника, столбец `MANAGER_EMP_ID` должен допускать использование NULL.

Для столбца `MANAGER_EMP_ID` существует ограничение внешнего ключа, накладывающее на него следующее правило: любое значение, помещаемое в данный столбец, должно представлять `EMP_ID` реального служащего. При работе с иерархической информацией такое ограничение применять не обязательно. Но в целом, задание ограничений для реализации бизнес-правил является хорошей привычкой.

Прежде чем перейти к разделам, посвященным обработке иерархических структур, необходимо ввести несколько понятий. Приведем список терминов, которые будут использоваться при работе с иерархическими данными:

#### *Узел (node)*

Строка таблицы, представляющая конкретную запись иерархической древовидной структуры. Например, на рис. 8.1 каждый служащий – это узел.

#### *Родитель (parent)*

Узел, находящийся в дереве одним уровнем выше. На рис. 8.1 King – это родитель для Blake, а Blake – родитель для Martin. Используется как термин *родительский узел*, так и просто *родитель*.

#### *Потомок (child)*

Узел, находящийся в дереве одним уровнем ниже. На рис. 8.1 Blake – это потомок King. Узел King, в свою очередь, имеет пять потомков:



Allen, Ward, Martin, Turner и James. Наряду с термином *потомок* используется термин *дочерний узел*.

### *Корень (root)*

Самый верхний узел иерархической структуры. Для корня не существует родителя. На рис. 8.1 корнем является узел King. В каждом дереве существует только один корень, но в иерархической таблице может содержаться несколько деревьев. Если таблица служащих хранит данные о сотрудниках нескольких компаний, в ней будет по одному корню для каждой компании. Используется как термин *корень*, так и *корневой узел*.

### *Лист (leaf)*

Узел, у которого нет потомков. Листья являются противоположностью корней и представляют собой нижний уровень древовидной структуры. На рис. 8.1 *концевые узлы (leaf node*, еще один термин для *листьев*) — это Adams, Smith, Allen, Ward, Martin, Turner, James и Miller. Листья не обязаны находиться все на одном уровне, они лишь не должны иметь дочерних узлов.

### *Уровень (level)*

Слой узлов. На рис. 8.1 King образует один уровень; Jones, Blake и Clark — следующий (более низкий) уровень и т. д.

## Простые операции над иерархическими данными

Процессы извлечения информации из таблицы, хранящей иерархические данные, достаточно просты, и для их осуществления достаточно примеров, уже изученных вами в этой книге. Для извлечения более сложной информации необходимы новые конструкции SQL, которые будут представлены в разделе «Расширения Oracle SQL» далее в этой главе. Пока же поговорим о тех операциях, для выполнения которых у вас уже достаточно знаний.

## Поиск корневого узла

Найти корень иерархической структуры очень просто — будем искать узел (единственный), у которого нет родителя. В рассматриваемой ранее таблице EMPLOYEE значение MANAGER\_EMP\_ID содержит NULL для самого вышестоящего служащего и только для него. Напишем запрос, который ищет NULL в столбце MANAGER\_EMP\_ID и, соответственно, возвращает корневой узел:

```
SELECT EMP_ID, LNAME, DEPT_ID, MANAGER_EMP_ID, SALARY, HIRE_DATE  
FROM EMPLOYEE  
WHERE MANAGER_EMP_ID IS NULL;
```

EMP_ID	LNAME	DEPT_ID	MANAGER_EMP_ID	SALARY	HIRE_DATE
7839	KING	10		5000	17-NOV-81

Так как иерархия структуры определяется столбцом MANAGER\_EMP\_ID, важно, чтобы в нем содержались только корректные данные. При заполнении этой таблицы необходимо проследить за тем, чтобы для каждой строки (кроме строки самого вышестоящего начальника) было указано значение для столбца MANAGER\_EMP\_ID. Самый главный служащий ни перед кем не отчитывается (у него нет руководителя), следовательно, MANAGER\_EMP\_ID для него не применим. Если не задать значения MANAGER\_EMP\_ID для сотрудников, у которых есть руководители, они будут ошибочно представлены корневыми узлами.

## Поиск непосредственного родителя узла

Может понадобиться связать узлы с их прямыми родителями. Например, нужно вывести в отчете руководителя каждого сотрудника. Фамилию руководителя каждого служащего можно получить, объединив таблицу EMPLOYEE с ней же самой. Такой тип объединения называется самообъединением (о них рассказывалось в главе 3). Следующий запрос возвращает нужный результат:

```
SELECT E.LNAME "Employee", M.LNAME "Manager"
FROM EMPLOYEE E, EMPLOYEE M
WHERE E.MANAGER_EMP_ID = M.EMP_ID;
```

Employee	Manager
SMITH	FORD
ALLEN	BLAKE
WARD	BLAKE
JONES	KING
MARTIN	BLAKE
BLAKE	KING
CLARK	KING
SCOTT	JONES
TURNER	BLAKE
ADAMS	SCOTT
JAMES	BLAKE
FORD	JONES
MILLER	CLARK

13 rows selected.

Заметьте, что запрос возвращает 13 строк, хотя в таблице EMPLOYEE 14 строк.

```
SELECT COUNT(*) FROM EMPLOYEE;
```

COUNT(*)
14

Причина того, что самообъединение возвращает всего 13 строк, проста. Запрос выводит служащих и их руководителей. А поскольку занимающий самое высокое положение служащий KING не имеет руководителя, соответствующая строка не попадает в результирующее множество. Если вы хотите, чтобы были выведены все сотрудники, необходимо использовать внешнее объединение:

```
SELECT E.LNAME "Employee", M.LNAME "Manager"
FROM EMPLOYEE E, EMPLOYEE M
WHERE E.MANAGER_EMP_ID = M.EMP_ID (+);
```

Employee	Manager
SMITH	FORD
ALLEN	BLAKE
WARD	BLAKE
JONES	KING
MARTIN	BLAKE
BLAKE	KING
CLARK	KING
SCOTT	JONES
KING	
TURNER	BLAKE
ADAMS	SCOTT
JAMES	BLAKE
FORD	JONES
MILLER	CLARK

14 rows selected.

Внешние объединения подробно рассмотрены в главе 3.

## Поиск концевых узлов

Задачей, противоположной поиску корневых узлов, является нахождение узлов-листьев, не имеющих дочерних узлов. Служащие, которые никем не руководят, – это концевые узлы иерархической древовидной структуры, представленной на рис. 8.1. На первый взгляд кажется, что запрос должен выводить всех сотрудников из таблицы EMPLOYEE, которые никем не руководят:

```
SELECT * FROM EMPLOYEE
WHERE EMP_ID NOT IN (SELECT MANAGER_EMP_ID FROM EMPLOYEE);
```

Однако если вы выполните этот оператор, будет выведено «No rows selected» (не выбрано ни одной строки). Почему? Дело в том, что столбец MANAGER\_EMP\_ID содержит в одной строке (для самого вышестоящего начальника) значение NULL, которое нельзя сравнивать с другими значениями. Поэтому, чтобы получить список сотрудников, которые никем не руководят, следует переформулировать запрос так:



```

SELECT EMP_ID, LNAME, DEPT_ID, MANAGER_EMP_ID, SALARY, HIRE_DATE
FROM EMPLOYEE E
WHERE EMP_ID NOT IN
(SELECT MANAGER_EMP_ID FROM EMPLOYEE
WHERE MANAGER_EMP_ID IS NOT NULL);

```

EMP_ID	LNAME	DEPT_ID	MANAGER_EMP_ID	SALARY	HIRE_DATE
7369	SMITH	20	7902	800	17-DEC-80
7499	ALLEN	30	7698	1600	20-FEB-81
7521	WARD	30	7698	1250	22-FEB-81
7654	MARTIN	30	7698	1250	28-SEP-81
7844	TURNER	30	7698	1500	08-SEP-81
7876	ADAMS	20	7788	1100	23-MAY-87
7900	JAMES	30	7698	950	03-DEC-81
7934	MILLER	10	7782	1300	23-JAN-82

8 rows selected.

В данном примере подзапрос возвращает значение EMP\_ID для всех руководителей. Затем внешний запрос возвращает всех служащих, кроме тех, которые были возвращены подзапросом. Используя EXISTS вместо IN, можно применить в операторе связанный подзапрос:

```

SELECT EMP_ID, LNAME, DEPT_ID, MANAGER_EMP_ID, SALARY, HIRE_DATE
FROM EMPLOYEE E
WHERE NOT EXISTS
(SELECT EMP_ID FROM EMPLOYEE E1 WHERE E.EMP_ID = E1.MANAGER_EMP_ID);

```

EMP_ID	LNAME	DEPT_ID	MANAGER_EMP_ID	SALARY	HIRE_DATE
7369	SMITH	20	7902	800	17-DEC-80
7499	ALLEN	30	7698	1600	20-FEB-81
7521	WARD	30	7698	1250	22-FEB-81
7654	MARTIN	30	7698	1250	28-SEP-81
7844	TURNER	30	7698	1500	08-SEP-81
7876	ADAMS	20	7788	1100	23-MAY-87
7900	JAMES	30	7698	950	03-DEC-81
7934	MILLER	10	7782	1300	23-JAN-82

8 rows selected.

Связанный подзапрос проверяет каждого служащего, чтобы посмотреть, является ли он чьим-нибудь руководителем. Если нет, то служащий включается в результирующее множество.

## Расширения Oracle SQL

В последних примерах вы увидели, как выполнить некоторые простые операции над иерархическими структурами посредством простых конструкций SQL. Другие операции, такие как обход дерева, выделение уровней и т. д., требуют более сложных операторов SQL, а также при-

менения функциональных возможностей, разработанных специально для работы с иерархическими данными. Для облегчения выполнения подобных операций Oracle предоставляет расширения ANSI SQL. Но прежде чем переходить к изучению расширений Oracle SQL, давайте посмотрим, как можно совершить обход дерева, используя ANSI SQL, и какие проблемы могут при этом возникнуть.

Пусть, например, вы хотите вывести всех сотрудников и их руководителей. Используя стандартный Oracle SQL, можно выполнить внешние самообъединения для таблицы EMPLOYEE:

```
SELECT E_TOP.LNAME, E_2.LNAME, E_3.LNAME, E_4.LNAME
FROM EMPLOYEE E_TOP, EMPLOYEE E_2, EMPLOYEE E_3, EMPLOYEE E_4
WHERE E_TOP.MANAGER_EMP_ID IS NULL
AND E_TOP.EMP_ID = E_2.MANAGER_EMP_ID (+)
AND E_2.EMP_ID = E_3.MANAGER_EMP_ID (+)
AND E_3.EMP_ID = E_4.MANAGER_EMP_ID (+);
```

LNAME	LNAME	LNAME	LNAME
KING	BLAKE	ALLEN	
KING	BLAKE	WARD	
KING	BLAKE	MARTIN	
KING	JONES	SCOTT	ADAMS
KING	BLAKE	TURNER	
KING	BLAKE	JAMES	
KING	JONES	FORD	SMITH
KING	CLARK	MILLER	

8 rows selected.

Запрос возвращает 8 строк, соответствующих восьми ветвям дерева. Чтобы получить такой результат, запрос выполняет самообъединение для четырех экземпляров таблицы EMPLOYEE. Четыре экземпляра таблицы EMPLOYEE необходимы этому оператору, потому что существует четыре уровня иерархии. Каждый уровень представлен одной таблицей EMPLOYEE. Внешнее объединение обусловлено тем, что для одного служащего (KING) столбец MANAGER\_EMP\_ID содержит NULL.

У такого запроса несколько недостатков. Прежде всего, для его написания необходимо знать количество уровней организационной структуры, а такие сведения не всегда доступны. Еще менее вероятно, что количество уровней навсегда останется неизменным. Кроме того, для четырех уровней необходимо объединить четыре экземпляра таблицы EMPLOYEE. А представьте себе организацию с двадцатью уровнями: придется объединить 20 таблиц. Это приведет к серьезным проблемам с производительностью.

Для обхода таких проблем Oracle предоставляет расширения ANSI SQL, а именно три конструкции, которые помогут рационально и эффективно выполнять иерархические запросы:

- Инструкция START WITH... CONNECT BY



- Оператор `PRIOR`
- Псевдостолбец `LEVEL`

В следующих разделах эти расширения Oracle будут изучены подробно.

## START WITH...CONNECT BY и PRIOR

Используя в операторе `SELECT` инструкцию `START WITH...CONNECT BY`, можно извлекать из таблицы, содержащей иерархические данные, информацию в иерархическом виде. Формат этой инструкции таков:

```
[[START WITH условие1] CONNECT BY условие2]
```

Рассмотрим элементы синтаксиса:

### *START WITH* условие1

Указывает корневую строку (строки) иерархии. Все строки, удовлетворяющие *условию1*, рассматриваются как корневые. Если не указывать инструкцию `START WITH`, все строки будут считаться корневыми. В *условии1* можно включать подзапрос.

### *CONNECT BY* условие2

Указывает отношение между родительскими и дочерними строками иерархии. Отношение выражается как сравнение, в котором столбцы текущей строки сравниваются с соответствующими столбцами родителя. *Условие2* должно содержать оператор `PRIOR`, который идентифицирует столбцы родительской строки. В *условии2* нельзя включать подзапрос.

`PRIOR` — это встроенный оператор Oracle SQL, который используется только для иерархических запросов. В иерархическом запросе инструкция `CONNECT BY` определяет отношение между родительской и дочерней строками. Когда в выражении условия `CONNECT BY` использован оператор `PRIOR`, выражение, следующее за ключевым словом `PRIOR`, вычисляется для родителя текущей строки запроса. В следующем примере `PRIOR` используется для связи каждой строки с родительской путем объединения столбца `MANAGER_EMP_ID` дочерней строки со столбцом `EMP_ID` родительской:

```
SELECT LNAME, EMP_ID, MANAGER_EMP_ID
FROM EMPLOYEE
START WITH MANAGER_EMP_ID IS NULL
CONNECT BY PRIOR EMP_ID = MANAGER_EMP_ID;
```

LNAME	EMP_ID	MANAGER_EMP_ID
KING	7839	
JONES	7566	7839
SCOTT	7788	7566
ADAMS	7876	7788
FORD	7902	7566



SMITH	7369	7902
BLAKE	7698	7839
ALLEN	7499	7698
WARD	7521	7698
MARTIN	7654	7698
TURNER	7844	7698
JAMES	7900	7698
CLARK	7782	7839
MILLER	7934	7782

14 rows selected.

Оператор PRIOR не обязательно должен указываться для первого столбца. Предыдущий запрос можно переписать и так:

```
SELECT LNAME, EMP_ID, MANAGER_EMP_ID
FROM EMPLOYEE
START WITH MANAGER_EMP_ID IS NULL
CONNECT BY MANAGER_EMP_ID = PRIOR EMP_ID;
```

Так как условие CONNECT BY задает отношение родитель-потомок, оно не может содержать петли. Если строка одновременно является для другой строки и родительской (прямым предком), и дочерней (прямым потомком), возникает петля. Например, петля бы возникла, если бы в таблицу EMPLOYEE входили две такие строки:

EMP_ID	LNAME	DEPT_ID	MANAGER_EMP_ID	SALARY	HIRE_DATE
9001	SMITH	20	9002	1800	15-NOV-61
9002	ALLEN	30	9001	11800	16-NOV-61

Когда отношение родитель-потомок затрагивает более двух столбцов, оператор PRIOR необходимо использовать перед каждым родительским столбцом. Давайте рассмотрим процесс сборки на заводе-изготовителе. Изделие может состоять из нескольких узлов, каждый из которых, в свою очередь, может состоять из нескольких деталей. Все они хранятся в таблице ASSEMBLY:

DESC ASSEMBLY		
Name	Null?	Type
ASSEMBLY_TYPE	NOT NULL	VARCHAR2(4)
ASSEMBLY_ID	NOT NULL	NUMBER(6)
DESCRIPTION	NOT NULL	VARCHAR2(20)
PARENT_ASSEMBLY_TYPE		VARCHAR2(4)
PARENT_ASSEMBLY_ID		NUMBER(6)

Столбцы ASSEMBLY\_TYPE и ASSEMBLY\_ID образуют первичный ключ этой таблицы, а столбцы PARENT\_ASSEMBLY\_TYPE и PARENT\_ASSEMBLY\_ID вместе составляют внешний ключ таблицы для себя самой. Следовательно, для применения к данной таблице иерархического запроса необходимо в

инструкции START WITH и CONNECT BY включить оба столбца. Кроме того, перед каждым родительским столбцом необходимо использовать оператор PRIOR:

```
SELECT * FROM ASSEMBLY
START WITH PARENT_ASSEMBLY_TYPE IS NULL
AND PARENT_ASSEMBLY_ID IS NULL
CONNECT BY PARENT_ASSEMBLY_TYPE = PRIOR ASSEMBLY_TYPE
AND PARENT_ASSEMBLY_ID = PRIOR ASSEMBLY_ID;
```

ASSE	ASSEMBLY_ID	DESCRIPTION	PARE	PARENT_ASSEMBLY_ID
A	1234	Assembly A#1234		
A	1256	Assembly A#1256	A	1234
B	6543	Part Unit#6543	A	1234
A	1675	Part Unit#1675	B	6543
X	9943	Repair Zone 1		
X	5438	Repair Unit #5438	X	9943
X	1675	Readymade Unit #1675	X	5438

7 rows selected.

## Псевдостолбец LEVEL

В иерархическом дереве термин *уровень (level)* подразумевает уровень узлов. Например, на рис. 8.1 корневой узел (представленный служащим KING) – это уровень 1. Следующий слой служащих (JONES, BLAKE, CLARK) – это уровень 2 и т. д. Для представления уровней иерархического дерева Oracle предоставляет псевдостолбец LEVEL. Каждый раз, когда в иерархическом запросе используются инструкции START WITH и CONNECT BY, вы можете использовать псевдостолбец LEVEL для получения номера уровня каждой строки, возвращенной запросом. Рассмотрим пример использования псевдостолбца LEVEL:

```
SELECT LEVEL, LNAME, EMP_ID, MANAGER_EMP_ID
FROM EMPLOYEE
START WITH MANAGER_EMP_ID IS NULL
CONNECT BY MANAGER_EMP_ID = PRIOR EMP_ID;
```

LEVEL	LNAME	EMP_ID	MANAGER_EMP_ID
1	KING	7839	
2	JONES	7566	7839
3	SCOTT	7788	7566
4	ADAMS	7876	7788
3	FORD	7902	7566
4	SMITH	7369	7902
2	BLAKE	7698	7839
3	ALLEN	7499	7698
3	WARD	7521	7698
3	MARTIN	7654	7698

3 TURNER	7844	7698
3 JAMES	7900	7698
2 CLARK	7782	7839
3 MILLER	7934	7782

14 rows selected.

Заметьте, что каждому служащему сопоставлен номер (псевдостолбец LEVEL), соответствующий его уровню в организационной структуре компании (см. рис. 8.1).

## Сложные иерархические операции

В этом разделе вы узнаете, как можно применять расширения Oracle SQL для выполнения сложных иерархических запросов.

### Вычисление количества уровней

Вы уже знаете, что при использовании инструкций START WITH и CONNECT BY псевдостолбец LEVEL генерирует номер уровня для каждой записи. Для определения количества уровней иерархической структуры считаем количество различных номеров, возвращенных псевдостолбцом LEVEL:

```
SELECT COUNT(DISTINCT LEVEL)
FROM EMPLOYEE
START WITH MANAGER_EMP_ID IS NULL
CONNECT BY PRIOR EMP_ID = MANAGER_EMP_ID;

COUNT(DISTINCT LEVEL)
-----
4
```

Чтобы определить количество служащих каждого уровня, сгруппируем результаты по столбцу LEVEL и подсчитаем количество сотрудников каждой группы, например:

```
SELECT LEVEL, COUNT(EMP_ID)
FROM EMPLOYEE
START WITH MANAGER_EMP_ID IS NULL
CONNECT BY PRIOR EMP_ID = MANAGER_EMP_ID
GROUP BY LEVEL;

LEVEL COUNT(EMP_ID)
-----
1          1
2          3
3          8
4          2
```



## Вывод записей в иерархическом порядке

Программисты, пишущие на SQL, очень часто сталкиваются с необходимостью вывода записей иерархической структуры в правильном иерархическом порядке. Например, может понадобиться вывести сотрудников, а под ними их подчиненных, как в следующем запросе:

```
SELECT LEVEL, LPAD(' ', 2*(LEVEL - 1)) || LNAME "EMPLOYEE",  
       EMP_ID, MANAGER_EMP_ID  
FROM EMPLOYEE  
START WITH MANAGER_EMP_ID IS NULL  
CONNECT BY PRIOR EMP_ID = MANAGER_EMP_ID;
```

LEVEL	Employee	EMP_ID	MANAGER_EMP_ID
1	KING	7839	
2	JONES	7566	7839
3	SCOTT	7788	7566
4	ADAMS	7876	7788
3	FORD	7902	7566
4	SMITH	7369	7902
2	BLAKE	7698	7839
3	ALLEN	7499	7698
3	WARD	7521	7698
3	MARTIN	7654	7698
3	TURNER	7844	7698
3	JAMES	7900	7698
2	CLARK	7782	7839
3	MILLER	7934	7782

14 rows selected.

Обратите внимание на использование выражения `LPAD(' ', 2*(LEVEL - 1))`, благодаря которому можно выровнять фамилии сотрудников согласно их уровням. При увеличении номера уровня увеличивается количество пробелов, возвращаемых выражением, и отступ перед фамилией служащего увеличивается.

Предыдущий запрос выводит всех служащих таблицы `EMPLOYEE`. Если вы хотите отфильтровать их по какому-то условию, можно использовать в нашем иерархическом запросе инструкцию `WHERE`, например:

```
SELECT LEVEL, LPAD(' ', 2*(LEVEL - 1)) || LNAME "EMPLOYEE",  
       EMP_ID, MANAGER_EMP_ID, SALARY  
FROM EMPLOYEE  
WHERE SALARY > 2000  
START WITH MANAGER_EMP_ID IS NULL  
CONNECT BY MANAGER_EMP_ID = PRIOR EMP_ID;
```

LEVEL	Employee	EMP_ID	MANAGER_EMP_ID	SALARY
1	KING	7839		5000

3	SCOTT	7788	7566	3000
3	FORD	7902	7566	3000
2	BLAKE	7698	7839	2850
2	CLARK	7782	7839	2450

Данный запрос возвращает фамилии служащих, зарплата которых больше 2000. Заметьте, что инструкция WHERE ограничивает множество строк, возвращаемых запросом, но никак не влияет на иерархию оставшихся строк. Например, служащий JONES был удален инструкцией WHERE из результирующего множества, но находившиеся в иерархии под ним сотрудники SCOTT и FORD не были отфильтрованы и продолжают занимать в выводе то же положение, что и в присутствии JONES (то есть отступ перед ними сохраняется неизменным). Инструкция WHERE в иерархическом запросе должна предшествовать инструкции START WITH...CONNECT BY, иначе будет выдано сообщение о синтаксической ошибке.

Вместо того чтобы выводить всю структуру организации, можно, например, вывести только поддерево, расположенное под определенным служащим, например JONES. Изменим условие в инструкции START WITH так, чтобы корнем запроса стал JONES, например:

```
SELECT LEVEL, LPAD(' ', 2*(LEVEL - 1)) || LNAME "EMPLOYEE",
       EMP_ID, MANAGER_EMP_ID, SALARY
FROM EMPLOYEE
START WITH LNAME = 'JONES'
CONNECT BY MANAGER_EMP_ID = PRIOR EMP_ID;
```

LEVEL	Employee	EMP_ID	MANAGER_EMP_ID	SALARY
1	JONES	7566	7839	2000
2	SCOTT	7788	7566	3000
3	ADAMS	7876	7788	1100
2	FORD	7902	7566	3000
3	SMITH	7369	7902	800

Заметьте, что, так как корнем иерархии стал JONES, запрос присваивает ему уровень 1, его непосредственным подчиненным – уровень 2 и т. д. Будьте внимательны при использовании в иерархических запросах условий, подобных LNAME = 'JONES'. Если бы в компании работали два сотрудника с такой фамилией, мог бы быть получен неверный результат. В подобных ситуациях предпочтительно использовать в условиях столбцы первичного или уникального ключа, такие как EMP\_ID.

В этом примере была выведена часть организационной схемы компании, возглавляемая определенным сотрудником. Может возникнуть необходимость вывести перечень служащих, возглавляемых сотрудником, удовлетворяющим некоторому условию. Например, можно вывести всех служащих, возглавляемых тем сотрудником, который дольше всех работает в компании. В данном случае отправная точка запроса (корень) будет зависеть от условия. Следовательно, необходи-



мо использовать подзапрос для получения информации и передачи ее основному запросу:

```
SELECT LEVEL, LPAD(' ', 2*(LEVEL - 1)) || LNAME "EMPLOYEE",
       EMP_ID, MANAGER_EMP_ID, SALARY
FROM EMPLOYEE
START WITH HIRE_DATE = (SELECT MIN(HIRE_DATE) FROM EMPLOYEE)
CONNECT BY MANAGER_EMP_ID = PRIOR EMP_ID;
```

LEVEL	EMPLOYEE	EMP_ID	MANAGER_EMP_ID	SALARY
1	BLAKE	7698	7839	2850
2	ALLEN	7499	7698	1600
2	WARD	7521	7698	1250
2	MARTIN	7654	7698	1250
2	TURNER	7844	7698	1500
2	JAMES	7900	7698	950

6 rows selected.

Обратите внимание на инструкцию START WITH. Подзапрос в инструкции START WITH возвращает минимальное значение столбца HIRE\_DATE (дата приема на работу) в таблице, то есть дату приема на работу «старейшего» служащего компании. Основной запрос использует эту информацию для определения начальной точки иерархии и выводит часть структуры организации, возглавляемую данным служащим.

Прежде чем использовать подзапрос в инструкции START WITH, подумайте, сколько строк им будет возвращено. Если будет возвращено несколько строк, в то время как ожидается одна (на что указывает знак «=»), будет выведено сообщение об ошибке. Можно обойти эту ситуацию, используя вместо «=» оператор IN, но знайте, что дело может закончиться тем, что в иерархическом запросе окажется несколько корней.

## Проверка подчиненности

Еще одной распространенной операцией над иерархическими структурами является проверка подчиненности. Зная схему организации, вы можете задать вопрос о том, имеет ли один служащий власть над другим, например, подчиняется ли служащий BLAKE служащему JONES. Чтобы ответить на такой вопрос, необходимо провести поиск служащего BLAKE в поддереве, возглавляемом служащим JONES. Если BLAKE найден в поддереве, это означает, что он прямо или косвенно подчиняется JONES. Если же BLAKE не обнаружен в поддереве, значит, никакой власти над ним JONES не имеет. Рассмотрим запрос, реализующий поиск BLAKE в поддереве, возглавляемом JONES:

```
SELECT *
FROM EMPLOYEE
WHERE LNAME = 'BLAKE'
START WITH LNAME = 'JONES'
```



```
CONNECT BY MANAGER_EMP_ID = PRIOR EMP_ID;
```

no rows selected

Инструкция `START WITH...CONNECT BY` генерирует поддерево, которым руководит JONES, а инструкция `WHERE` фильтрует это поддерево и ищет служащего BLAKE. Как видите, запрос не возвращает строк, то есть BLAKE не был найден в поддереве, возглавляемом JONES, и теперь вы знаете, что BLAKE ему не подчиняется. Давайте рассмотрим другой пример, который выдает положительный результат. Проверим, подчиняется ли SMITH служащему JONES:

```
SELECT EMP_ID, LNAME, DEPT_ID, MANAGER_EMP_ID, SALARY, HIRE_DATE
FROM EMPLOYEE
WHERE LNAME = 'SMITH'
START WITH LNAME = 'JONES'
CONNECT BY MANAGER_EMP_ID = PRIOR EMP_ID;
```

EMP_ID	LNAME	DEPT_ID	MANAGER_EMP_ID	SALARY	HIRE_DATE
7369	SMITH	20	7902	800	17-DEC-80

На этот раз SMITH был найден в списке сотрудников, руководимых JONES, и теперь вы знаете, что JONES является начальником некоторого уровня для сотрудника SMITH.

## Удаление поддерева

Давайте предположим, что организация, с которой мы работаем, разделяется, и JONES со всеми своими подчиненными образует новую компанию. Значит, больше не нужно хранить данные об этих служащих в таблице EMPLOYEE. Необходимо удалить из таблицы целое поддерево, возглавляемое сотрудником JONES (см. рис. 8.1). Сделаем это, используя подзапрос, как показано в следующем примере:

```
DELETE FROM EMPLOYEE
WHERE EMP_ID IN
(SELECT EMP_ID FROM EMPLOYEE
START WITH LNAME = 'JONES'
CONNECT BY MANAGER_EMP_ID = PRIOR EMP_ID);
```

5 rows deleted.

Подзапрос генерирует поддерево, возглавляемое JONES, и возвращает значения EMP\_ID для всех служащих этого поддерева, включая самого JONES. Затем внешний запрос удаляет записи с возвращенными значениями EMP\_ID из таблицы EMPLOYEE.

## Вывод нескольких корневых узлов

Интересным вариантом задачи нахождения корневого узла иерархии является поиск и вывод корневых узлов нескольких иерархий, храня-

щихся в одной таблице. Можно, например, считать корневыми узлами руководителей подразделений и решить задачу вывода списка всех руководителей подразделений из таблицы EMPLOYEE.

На служащих какого бы то ни было подразделения не налагается никаких ограничений. Однако можно считать, что если А подчиняется В и В подчиняется С, и А и С относятся к одному подразделению, то В относится к тому же подразделению. Если руководитель служащего относится к другому подразделению, это означает, что этот служащий занимает наивысшее положение в своем подразделении, то есть является его руководителем.

Следовательно, чтобы найти руководителей подразделений, нужно искать служащих, руководители которых относятся не к тому подразделению, к которому принадлежит сам служащий. Напишем такой запрос:

```
SELECT EMP_ID, LNAME, DEPT_ID, MANAGER_EMP_ID, SALARY, HIRE_DATE
FROM EMPLOYEE
START WITH MANAGER_EMP_ID IS NULL
CONNECT BY MANAGER_EMP_ID = PRIOR EMP_ID
AND DEPT_ID != PRIOR DEPT_ID;
```

EMP_ID	LNAME	DEPT_ID	MANAGER_EMP_ID	SALARY	HIRE_DATE
7839	KING	10		5000	17-NOV-81
7566	JONES	20	7839	2975	02-APR-81
7698	BLAKE	30	7839	2850	01-MAY-81

Дополнительное условие DEPT\_ID != PRIOR DEPT\_ID, добавленное в инструкцию CONNECT BY, ограничивает вывод теми служащими, руководители которых относятся к другому подразделению.

## Вывод нескольких верхних уровней иерархии

Еще одной задачей, часто встречающейся при работе с иерархическими данными, является вывод нескольких верхних уровней дерева. Например, может потребоваться вывести список высшего исполнительного руководства компании. Пусть в нашей организации к высшей администрации относятся два верхних уровня. Используем для идентификации таких служащих псевдостолбец LEVEL:

```
SELECT EMP_ID, LNAME, DEPT_ID, MANAGER_EMP_ID, SALARY, HIRE_DATE
FROM EMPLOYEE
WHERE LEVEL <= 2
START WITH MANAGER_EMP_ID IS NULL
CONNECT BY MANAGER_EMP_ID = PRIOR EMP_ID;
```

EMP_ID	LNAME	DEPT_ID	MANAGER_EMP_ID	SALARY	HIRE_DATE
7839	KING	10		5000	17-NOV-81



7566 JONES	20	7839	2000 02-APR-81
7698 BLAKE	30	7839	2850 01-MAY-80
7782 CLARK	10	7839	2450 09-JUN-81

В этом примере условие `LEVEL <= 2` инструкции `WHERE` ограничивает результирующее множество служащими двух верхних уровней организационной структуры.

## Обобщение иерархических данных

Еще одна интересная операция над иерархическими структурами – это обобщение иерархических данных. Например, может потребоваться просуммировать зарплаты сотрудников, подчиняющихся какому-то служащему. Или же можно рассматривать каждого служащего как корень и выводить суммы зарплат всех подчиняющихся ему сотрудников.

Первая задача достаточно проста. Вы уже знаете, как выбрать поддерево, возглавляемое определенным служащим. Просуммировать зарплаты всех служащих такого поддерева также не составляет труда:

```
SELECT SUM(SALARY)
FROM EMPLOYEE
START WITH LNAME = 'JONES'
CONNECT BY MANAGER_EMP_ID = PRIOR EMP_ID;
```

```
SUM(SALARY)
-----
      10875
```

Инструкция `START WITH LNAME = 'JONES'` генерирует поддерево, возглавляемое сотрудником **JONES**, а выражение `SUM(SALARY)` суммирует зарплаты всех служащих поддерева.

Вторая задача, казалось бы, является простым расширением первой, но на самом деле достаточно сложна. Нужно интерпретировать каждого сотрудника как корень и суммировать зарплаты всех сотрудников его поддерева. По существу, необходимо повторить только что написанный запрос для всех служащих компании. Используем встроенное представление:

```
SELECT LNAME, SALARY,
(SELECT SUM(SALARY) FROM EMPLOYEE T1
START WITH LNAME = T2.LNAME
CONNECT BY MANAGER_EMP_ID = PRIOR EMP_ID) SUM_SALARY
FROM EMPLOYEE T2;
```

LNAME	SALARY	SUM_SALARY
SMITH	800	800
ALLEN	1600	1600
WARD	1250	1250
JONES	2975	9900



MARTIN	1250	1250
BLAKE	2850	9400
CLARK	2450	3750
SCOTT	3000	4100
KING	5000	28050
TURNER	1500	1500
ADAMS	1100	1100
JAMES	950	950
FORD	3000	3800
MILLER	1300	1300

14 rows selected.

В данном примере инструкция `START WITH...CONNECT BY` встроенного представления генерирует поддереву для каждого служащего. Встроенное представление выполняется один раз для каждой строки таблицы `EMPLOYEE`. Для каждой строки этой таблицы оно генерирует поддерево, возглавляемое данным служащим, и возвращает сумму зарплат всех сотрудников поддерева в основной запрос.

Результирующее множество отображает два числа для каждого служащего. Первое число, `SALARY`, — это собственная зарплата сотрудника. Второе число, `SUM_SALARY`, — сумма зарплат всех сотрудников, подчиняющихся ему (включая его самого). Для решения подобных задач программисты часто прибегают к `PL/SQL`. Однако оператор, сочетающий мощь иерархических запросов и встроенных представлений, решает нашу задачу гораздо более элегантно и быстрым способом.

## Ограничения, налагаемые на иерархические запросы

На иерархические запросы, использующие инструкцию `START WITH...CONNECT BY`, налагаются следующие ограничения:

1. Иерархический запрос не может использовать объединение.



Существуют способы обойти это ограничение. В разделе «Расширения Oracle SQL» главы 5 был описан такой пример.

2. Иерархический запрос не может выбирать данные из представления, содержащего объединение.
3. В иерархическом запросе можно использовать инструкцию `ORDER BY`, но помните, что она имеет более высокий приоритет, чем классификация иерархии при помощи инструкции `START WITH...CONNECT BY`. Поэтому, если вас интересует только номер уровня, не имеет смысла применять инструкцию `ORDER BY` в иерархическом запросе.

Третий пункт требует небольшого пояснения. Давайте посмотрим на примере, что произойдет, если использовать инструкцию ORDER BY в иерархическом запросе:

```
SELECT LEVEL, LPAD(' ', 2*(LEVEL - 1)) || LNAME "EMPLOYEE",
       EMP_ID, MANAGER_EMP_ID, SALARY
FROM EMPLOYEE
START WITH MANAGER_EMP_ID IS NULL
CONNECT BY MANAGER_EMP_ID = PRIOR EMP_ID
ORDER BY SALARY;
```

LEVEL	Employee	EMP_ID	MANAGER_EMP_ID	SALARY
4	SMITH	7369	7902	800
3	JAMES	7900	7698	950
4	ADAMS	7876	7788	1100
3	WARD	7521	7698	1250
3	MARTIN	7654	7698	1250
3	MILLER	7934	7782	1300
3	TURNER	7844	7698	1500
3	ALLEN	7499	7698	1600
2	JONES	7566	7839	2000
2	CLARK	7782	7839	2450
2	BLAKE	7698	7839	2850
3	SCOTT	7788	7566	3000
3	FORD	7902	7566	3000
1	KING	7839		5000

14 rows selected.

Инструкция START WITH...CONNECT BY упорядочивает служащих согласно их истинному иерархическому порядку, однако поскольку в операторе есть инструкция ORDER BY с более высоким приоритетом, она упорядочивает сотрудников по их зарплате, деформируя иерархическое представление.

# 9

## DECODE и CASE

Будь то извлечение данных для представления пользователю, составление отчета или проведение расчетов, информация редко отображается именно в том виде, в котором она хранилась в базе данных. Данные комбинируются, преобразуются или форматируются. Процедурные языки, такие как PL/SQL и Java, предоставляют множество средств для работы с данными, но часто бывает желательно выполнять такие операции во время извлечения данных из базы. Аналогично при обновлении данных гораздо легче изменить их в процессе внесения, вместо того чтобы извлекать, изменять и возвращать обратно в базу данных. Эта глава будет посвящена двум мощным средствам Oracle SQL, обеспечивающим различные манипуляции с данными: выражению CASE и функции DECODE. Попутно будет описано применение некоторых других функций, таких как NVL и NVL2.

### DECODE, NVL и NVL2

Большинство встроенных функций Oracle создано для решения конкретных задач. Например, если вам необходимо найти последний день месяца, содержащего определенную дату, то функция LAST\_DAY – как раз то, что нужно. Однако функции DECODE, NVL и NVL2 не решают конкретных задач, их правильнее рассматривать как встроенные операторы if-then-else. Данные функции используются для принятия решений на основе значений данных в рамках оператора SQL, без обращения к процедурным языкам, таким как PL/SQL. Синтаксис и эквивалентная логическая конструкция для каждой из трех функций приведены в табл. 9.1.



Таблица 9.1. Логика функций в терминах if-then-else

Синтаксис функции	Логический эквивалент
DECODE(E1, E2, E3, E4)	IF E1 = E2 THEN E3 ELSE E4
NVL(E1, E2)	IF E1 IS NULL THEN E2 ELSE E1
NVL2(E1, E2, E3)	IF E1 IS NULL THEN E3 ELSE E2

В последующих двух разделах будет подробно рассказано о представленных функциях.

## DECODE

Функция DECODE может рассматриваться как встроенный оператор IF. DECODE принимает в качестве аргументов четыре или более выражений. Каждое из выражений может быть столбцом, литералом, функцией и даже подзапросом. Давайте рассмотрим простой пример использования DECODE:

```
SELECT lname,
       DECODE(manager_emp_id, NULL, 'MANAGER', 'NON-MANAGER') emp_type
FROM employee;
```

LNAME	EMP_TYPE
Brown	MANAGER
Smith	MANAGER
Blake	MANAGER
Freeman	NON-MANAGER
Grossman	NON-MANAGER
Thomas	NON-MANAGER
Powers	NON-MANAGER
Jones	NON-MANAGER
Levitz	NON-MANAGER
Boorman	NON-MANAGER
Fletcher	NON-MANAGER
Dunn	NON-MANAGER
Evans	NON-MANAGER
Walters	NON-MANAGER
Young	NON-MANAGER
Houseman	NON-MANAGER
McGowan	NON-MANAGER
Isaacs	NON-MANAGER
Jacobs	NON-MANAGER
King	NON-MANAGER
Fox	NON-MANAGER
Anderson	NON-MANAGER
Nichols	NON-MANAGER
Iverson	NON-MANAGER
Peters	NON-MANAGER
Russell	NON-MANAGER

В данном случае первое выражение – это столбец, второе – NULL, а третье и четвертое – символьные литералы. Задача заключалась в том, чтобы определить, является ли каждый из служащих руководителем: проверялось, содержит ли столбец `manager_emp_id` значение NULL. Функция `DECODE` сравнивает значение столбца `manager_emp_id` каждой строки (первое выражение) с NULL (второе выражение). Если результат выполнения сравнения – истина, то `DECODE` возвращает 'MANAGER' (третье выражение), иначе – 'NON-MANAGER' (последнее выражение).

Так как функция `DECODE` сравнивает два выражения и возвращает одно из двух выражений, важно, чтобы типы выражений совпадали или, по крайней мере, могли быть преобразованы в один и тот же тип. Приведенный ранее пример работает потому, что E1 можно сравнивать с E2, а E3 и E4 имеют одинаковые типы. Иначе бы Oracle вывел сообщение об ошибке:

```
SELECT lname,  
       DECODE(manager_emp_id, SYSDATE, 'MANAGER', 'NON-MANAGER') emp_type  
FROM employee;  
  
ERROR at line 1:  
ORA-00932: inconsistent datatypes
```

Числовой столбец `manager_emp_id` не может быть преобразован в тип `DATE`, поэтому сервер Oracle не в состоянии выполнить сравнение и генерирует исключение. Такое же исключение будет выдано, если два возвращаемых выражения (E3 и E4) будут иметь несравнимые типы.

В рассмотренном примере функция `DECODE` использовалась с минимальным количеством параметров – 4. Посмотрим, как можно использовать дополнительные параметры для задания более сложной логики:

```
SELECT p.part_nbr part_nbr, p.name part_name, s.name supplier,  
       DECODE(p.status, 'INSTOCK', 'In Stock',  
              'DISC', 'Discontinued',  
              'BACKORD', 'Backordered',  
              'ENROUTE', 'Arriving Shortly',  
              'UNAVAIL', 'No Shipment Scheduled',  
              'Unknown') part_status  
FROM part p, supplier s  
WHERE p.supplier_id = s.supplier_id;
```

Значение столбца статуса детали сравнивается с каждым из пяти значений, и, если найдено совпадение, выводится соответствующая строка. Если совпадение не найдено, возвращается строка 'Unknown'.

## NVL и NVL2

Функции `NVL` и `NVL2` позволяют проверить выражение на NULL. Если выражение равно NULL, можно вернуть другое, отличное от NULL значение, которое будет использоваться вместо него. Так как любое из выраже-

ний оператора DECODE может быть NULL, функции NVL и NVL2 фактически являются специализированными версиями DECODE. В следующем примере применение NVL2 приводит к выводу тех же результатов, что и при использовании DECODE (см. предыдущий раздел):

```
SELECT lname,
       NVL2(manager_emp_id, 'NON-MANAGER', 'MANAGER') emp_type
FROM employee;
```

LNAME	EMP_TYPE
Brown	MANAGER
Smith	MANAGER
Blake	MANAGER
Freeman	NON-MANAGER
Grossman	NON-MANAGER
Thomas	NON-MANAGER
Powers	NON-MANAGER
Jones	NON-MANAGER
Levitz	NON-MANAGER
Boorman	NON-MANAGER
Fletcher	NON-MANAGER
Dunn	NON-MANAGER
Evans	NON-MANAGER
Walters	NON-MANAGER
Young	NON-MANAGER
Houseman	NON-MANAGER
McGowan	NON-MANAGER
Isaacs	NON-MANAGER
Jacobs	NON-MANAGER
King	NON-MANAGER
Fox	NON-MANAGER
Anderson	NON-MANAGER
Nichols	NON-MANAGER
Iverson	NON-MANAGER
Peters	NON-MANAGER
Russell	NON-MANAGER

Функция NVL2 анализирует первое выражение, в данном случае — manager\_emp\_id. Если это выражение — NULL, NVL2 возвращает третье выражение. Если первое выражение — не NULL, NVL2 возвращает второе выражение. Используйте NVL2 при необходимости указать различные выводимые значения в тех случаях, когда выражение равно NULL и когда оно отлично от NULL.

Функция NVL обычно применяется для подстановки значения по умолчанию вместо NULL. В остальных случаях она возвращает само значение столбца. В следующем примере выводится идентификатор руководителя каждого служащего; для тех же служащих, которым не сопоставлен руководитель (то есть manager\_emp\_id содержит NULL), подставляется слово 'NONE':



```
SELECT emp.lname employee, NVL(mgr.lname, 'NONE') manager
FROM employee emp, employee mgr
WHERE emp.manager_emp_id = mgr.emp_id (+);
```

EMPLOYEE	MANAG
-----	-----
Brown	NONE
Smith	NONE
Blake	NONE
Freeman	Blake
Grossman	Blake
Thomas	Blake
Powers	Blake
Jones	Blake
Levitz	Blake
Boorman	Blake
Fletcher	Blake
Dunn	Blake
Evans	Blake
Walters	Blake
Young	Blake
Houseman	Blake
McGowan	Blake
Isaacs	Blake
Jacobs	Blake
King	Blake
Fox	King
Anderson	King
Nichols	King
Iverson	King
Peters	King
Russell	King

Несмотря на то что функция DECODE может заменить функции NVL и NVL2, большинство программистов для проверки выражения на NULL предпочитает использовать NVL или NVL2, вероятно потому, что ее значение интуитивно понятнее. Будем надеяться, что прочитав следующий раздел, вы убедитесь в том, что всякий раз, когда вам необходима функциональность оператора if-then-else, следует использовать выражение CASE. Тогда вам не придется беспокоиться о том, какую из встроенных функций следует использовать.

## История CASE

Выражение CASE появилось в спецификации SQL-92 в 1992 году. Восемь лет спустя Oracle включила выражение CASE в версию 8.1.6. Как и функция DECODE, выражение CASE включает в оператор SQL условную логику, чем можно объяснить, почему Oracle потратила столько времени на реализацию этой возможности. Если вы уже много лет работаете с Oracle, вас может удивить идея перехода на использование выра-

жения CASE в то время, когда со всем отлично справляется DECODE. Приведем причины, которые могут побудить вас изменить свое мнение:

- Выражения CASE могут использоваться везде, где разрешены функции DECODE.
- Выражения CASE более удобочитаемы, чем выражения DECODE.
- Выражения CASE выполняются быстрее, чем выражения DECODE.<sup>1</sup>
- Выражения CASE обрабатывают сложную логику более изящно, чем выражения DECODE.
- CASE соответствует стандарту ANSI, а DECODE представляет собой особенность Oracle SQL.

Единственным минусом использования CASE вместо DECODE является то, что выражения CASE не поддерживаются в Oracle8i PL/SQL. Если же вы работаете с Oracle9i, любые операторы SQL, выполняемые из PL/SQL, могут содержать выражения CASE.

Спецификация SQL-92 определяет два различных вида выражения CASE: *с поиском (searched)* и *простое (simple)*. В Oracle8i поддерживаются только выражения CASE с поиском. Если же вы работаете с Oracle9i, то можете использовать и простые выражения CASE.

## Выражения CASE с поиском

Выражение CASE с поиском вычисляет некоторое количество условий и возвращает результат, определяемый тем, какое из условий оказалось истинным. Синтаксис выражения CASE с поиском таков:

```
CASE
  WHEN C1 THEN R1
  WHEN C2 THEN R2
  ...
  WHEN CN THEN RN
  ELSE RD
END
```

В определении синтаксиса «С» представляют условия, а «R» – результаты. В каждом выражении CASE можно использовать до 127 инструкций WHEN, так что логика может быть достаточно сложной. Условия вычисляются по порядку. Когда найдено условие, оцененное как TRUE, возвращается соответствующий результат и выполнение CASE завершается. Чтобы обеспечить достижение желаемого результата, будьте внимательны при выборе порядка инструкций WHEN. Используем выра-

---

<sup>1</sup> Так как CASE встроено в грамматику Oracle SQL, нет необходимости в вызове функции для оценки логики *if-then-else*. Для одного вызова разница во времени исполнения будет мизерной, но при работе с большими результирующими множествами суммарная экономия времени на вызове функции станет значительной.



жение CASE для определения того, какая информация должна попасть в отчет о состоянии заказов:

```
SELECT co.order_nbr, co.cust_nbr,  
CASE WHEN co.expected_ship_dt IS NULL THEN 'NOT YET SCHEDULED'  
      WHEN co.expected_ship_dt <= SYSDATE THEN 'SHIPPING DELAYED'  
      WHEN co.expected_ship_dt <= SYSDATE + 2 THEN 'SHIPPING SOON'  
      ELSE 'BACKORDERED'  
END ship_status  
FROM cust_order co  
WHERE co.ship_dt IS NULL AND co.cancelled_dt IS NULL;
```

Подобно DECODE, все результаты выражения CASE должны иметь сопоставимые типы, иначе будет выдано сообщение об ошибке ORA-932. Каждое условие каждой инструкции WHEN не зависит от остальных, поэтому условия могут включать различные типы данных, как показано в следующем примере:

```
SELECT co.order_nbr, co.cust_nbr,  
CASE  
  WHEN co.sale_price > 10000 THEN 'BIG ORDER'  
  WHEN co.cust_nbr IN  
    (SELECT cust_nbr FROM customer WHERE tot_orders > 100)  
    THEN 'ORDER FROM FREQUENT CUSTOMER'  
  WHEN co.order_dt < TRUNC(SYSDATE) - 7 THEN 'OLD ORDER'  
  ELSE 'UNINTERESTING ORDER'  
END  
FROM cust_order co  
WHERE co.ship_dt IS NULL AND co.cancelled_dt IS NULL;
```

## Простые выражения CASE

Простые выражения CASE имеют структуру, отличную от выражений CASE с поиском. В них инструкции WHEN содержат не условия, а выражения, а в инструкцию CASE помещается единственное выражение, которое сравнивается с выражениями всех инструкций WHEN:

```
CASE E0  
  WHEN E1 THEN R1  
  WHEN E2 THEN R2  
  ...  
  WHEN EN THEN RN  
  ELSE RD  
END
```

Каждое из выражений E1...EN сравнивается с E0. Если найдено совпадение, возвращается соответствующий результат, иначе возвращается результат по умолчанию (RD). Все выражения должны быть одного типа, так как все они должны быть сравнимы с E0, что делает простые выражения CASE менее гибкими, чем выражения CASE с поиском. Применим простое выражение CASE для интерпретации кода статуса, хранящегося в таблице деталей:



```

SELECT p.part_nbr part_nbr, p.name part_name, s.name supplier,
CASE p.status
  WHEN 'INSTOCK' THEN 'In Stock'
  WHEN 'DISC' THEN 'Discontinued'
  WHEN 'BACKORD' THEN 'Backordered'
  WHEN 'ENROUTE' THEN 'Arriving Shortly'
  WHEN 'UNAVAIL' THEN 'No Shipment Scheduled'
  ELSE 'Unknown'
END part_status
FROM part p, supplier s
WHERE p.supplier_id = s.supplier_id;

```

Выражение CASE с поиском умеет делать все, что делает простое выражение CASE, что, вероятно, и является причиной того, что сначала в Oracle было реализовано только выражение CASE с поиском. Для некоторых задач, таких как интерпретация значений столбца, простые выражения могут оказаться более эффективными, если оцениваемое выражение вычисляется посредством вызова функции.

## Примеры использования DECODE и CASE

В последующих разделах будут представлены разнообразные примеры использования условной логики в операторах SQL. Несмотря на то что мы продолжаем рекомендовать везде, где это возможно, использовать выражение CASE, а не функцию DECODE, предлагается две версии (с использованием DECODE и с использованием CASE), чтобы вы могли проанализировать различия этих двух подходов.

## Преобразования результирующего множества

Может возникнуть ситуация, когда вы выполнили суммирование для некоторого конечного множества значений, например дней недели или месяцев года, и хотите, чтобы результирующее множество содержало одну строку из N столбцов, а не N строк с двумя столбцами. Рассмотрим запрос, который выполняет обобщение информации об объемах продаж за каждый квартал 2001 года:

```

SELECT TO_CHAR(order_dt, 'Q') sales_quarter,
SUM(sale_price) tot_sales
FROM cust_order
WHERE order_dt >= TO_DATE('01-JAN-2001', 'DD-MON-YYYY')
AND order_dt < TO_DATE('01-JAN-2002', 'DD-MON-YYYY')
GROUP BY TO_CHAR(order_dt, 'Q')
ORDER BY 1;

```

```

S  TOT_SALES
-----
1   9739328
2  10379833
3   9703114
4   9772633

```

Чтобы превратить результирующее множество в одну строку с четырьмя столбцами, необходимо создать столбец для каждого квартала и в каждом столбце просуммировать только те записи, для которых дата заказа относится к рассматриваемому кварталу. Воспользуемся функцией DECODE:

```
SELECT
  SUM(DECODE(TO_CHAR (order_dt, 'Q'), '1', sale_price, 0)) Q_1,
  SUM(DECODE(TO_CHAR (order_dt, 'Q'), '2', sale_price, 0)) Q_2,
  SUM(DECODE(TO_CHAR (order_dt, 'Q'), '3', sale_price, 0)) Q_3,
  SUM(DECODE(TO_CHAR (order_dt, 'Q'), '4', sale_price, 0)) Q_4
FROM cust_order
WHERE order_dt >= TO_DATE('01-JAN-2001', 'DD-MON-YYYY')
      AND order_dt < TO_DATE('01-JAN-2002', 'DD-MON-YYYY');
```

Q_1	Q_2	Q_3	Q_4
9739328	10379833	9703114	9772633

Все столбцы запроса практически идентичны, разница только в квартале, проверяемом функцией DECODE. Например, для столбца Q\_1 значение 0 возвращается до тех пор, пока заказы не попадают в первый квартал, в случае же попадания возвращается значение из столбца sale\_price. Хотя суммируются значения всех заказов 2001 года, к итоговому значению (для Q\_1) добавляются только заказы, сделанные в первом квартале, то есть фактически происходит суммирование всех заказов первого квартала без учета заказов, сделанных во 2, 3 и 4 кварталах. Та же самая логика используется и для столбцов Q\_2, Q\_3 и Q\_4, в которых суммируются заказы кварталов 2, 3 и 4, соответственно.

CASE-версия этого запроса такова:

```
SELECT
  SUM(CASE WHEN TO_CHAR(order_dt, 'Q') = '1' THEN sale_price ELSE 0 END) Q_1,
  SUM(CASE WHEN TO_CHAR(order_dt, 'Q') = '2' THEN sale_price ELSE 0 END) Q_2,
  SUM(CASE WHEN TO_CHAR(order_dt, 'Q') = '3' THEN sale_price ELSE 0 END) Q_3,
  SUM(CASE WHEN TO_CHAR(order_dt, 'Q') = '4' THEN sale_price ELSE 0 END) Q_4
FROM cust_order
WHERE order_dt >= TO_DATE('01-JAN-2001', 'DD-MON-YYYY')
      AND order_dt < TO_DATE('01-JAN-2002', 'DD-MON-YYYY');
```

Q_1	Q_2	Q_3	Q_4
9739328	10379833	9703114	9772633

Очевидно, что такие преобразования разумны, только если количество значений не очень велико. Обобщение продаж по кварталам или месяцам работает хорошо, но если вы решите расширить запрос, суммируя продажи по неделям, и завести по столбцу для каждой недели, это превратится в скучное и утомительное занятие.



## Выборочное выполнение функции

Предположим, что вам нужно составить опись товаров. Большая часть информации присутствует в вашей локальной базе данных, но, чтобы собрать информацию о деталях, поставленных Acme Industries, необходимо через шлюз обратиться к внешней базе, написанной не на Oracle. Запрос от вашей базы данных через шлюз к внешнему серверу и возврат результата в среднем занимает около 1,5 секунд. В вашей базе данных 10 000 деталей, но только 100 из них требуют получения внешней информации. Вы пишете пользовательскую функцию `get_resupply_date` для извлечения даты повторной поставки деталей ACME и включаете ее в запрос:

```
SELECT s.name supplier_name, p.name part_name, p.part_nbr part_number,
       p.inventory_qty in_stock, p.resupply_date resupply_date,
       my_pkg.get_resupply_date(p.part_nbr) acme_resupply_date
FROM part p, supplier s
WHERE p.supplier_id = s.supplier_id;
```

Затем вы включаете в вашу отчетную программу условную логику, чтобы использовать `acme_resupply_date` вместо столбца `resupply_date` в том случае, когда название поставщика – Acme Industries. Вы запускаете отчет, садитесь и ждете результатов. Ждете. Ждете...

К сожалению, сервер вынужден выполнить 10 000 обращений через шлюз, хотя необходимы всего 100. В таких ситуациях будет гораздо эффективнее вызывать функцию только тогда, когда это необходимо, вместо того чтобы вызывать ее всегда, а затем отбрасывать ненужные результаты:

```
SELECT s.name supplier_name, p.name part_name, p.part_nbr part_number,
       p.inventory_qty in_stock,
       DECODE(s.name, 'Acme Industries',
             my_pkg.get_resupply_date(p.part_nbr),
             p.resupply_date) resupply_date
FROM part p, supplier s
WHERE p.supplier_id = s.supplier_id;
```

Функция `DECODE` проверяет, совпадает ли имя поставщика с 'Acme Industries'. Если совпадает, вызывается функция, извлекающая дату повторной поставки через шлюз, иначе дата берется из локальной таблицы. CASE-версия этого запроса такова:

```
SELECT s.name supplier_name, p.name part_name, p.part_nbr part_number,
       p.inventory_qty in_stock,
       CASE WHEN s.name = 'Acme Industries'
            THEN my_pkg.get_resupply_date(p.part_nbr)
            ELSE p.resupply_date
       END resupply_date
FROM part p, supplier s
WHERE p.supplier_id = s.supplier_id;
```



Теперь пользовательская функция выполняется, только если поставщик – Асме, что радикально уменьшает время выполнения запроса. Дополнительную информацию о вызове пользовательских функций из SQL вы найдете в главе 11.

## Условное обновление

Если в процессе проектирования ваша база данных была денормализована, вы можете на ночь запускать утилиты для заполнения денормализованных столбцов. Например, таблица деталей содержит денормализованный столбец статуса, значение которого порождается столбцами `inventory_qty` и `resupply_date`. Чтобы обновить столбец статусов, можно каждую ночь выполнять четыре отдельных оператора `UPDATE`, по одному для каждого из четырех возможных значений столбца `status`:

```
UPDATE part SET status = 'INSTOCK'
WHERE inventory_qty > 0;

UPDATE part SET status = 'ENROUTE'
WHERE inventory_qty = 0 AND resupply_date < SYSDATE + 5;

UPDATE part SET status = 'BACKORD'
WHERE inventory_qty = 0 AND resupply_date > SYSDATE + 5;

UPDATE part SET status = 'UNAVAIL'
WHERE inventory_qty = 0 and resupply_date IS NULL;
```

Маловероятно, чтобы столбцы `inventory_qty` и `resupply_date` были индексированы, так что каждый из операторов `UPDATE` будет осуществлять полный просмотр таблицы деталей. Если же добавить в оператор условные выражения, можно скомбинировать четыре оператора `UPDATE` так, чтобы таблица `part` просматривалась всего единожды:

```
UPDATE part SET status =
  DECODE(inventory_qty, 0,
    DECODE(resupply_date, NULL, 'UNAVAIL',
      DECODE(LEAST(resupply_date, SYSDATE + 5), resupply_date,
        'ENROUTE', 'BACKORD')),
    'INSTOCK');
```

**CASE-версия этого оператора `UPDATE` такова:**

```
UPDATE part SET status =
  CASE WHEN inventory_qty > 0 THEN 'INSTOCK'
    WHEN resupply_date IS NULL THEN 'UNAVAIL'
    WHEN resupply_date < SYSDATE + 5 THEN 'ENROUTE'
    WHEN resupply_date > SYSDATE + 5 THEN 'BACKORD'
    ELSE 'UNKNOWN' END;
```

Выигрыш `CASE`-версии в читабельности очевиден: версия с функцией `DECODE` требует трех вложенных уровней для реализации той же условной логики, которую обрабатывает одно выражение `CASE`.

## Необязательное обновление

Иногда может понадобиться изменять данные только в определенных обстоятельствах. Например, пусть у вас есть таблица, в которой хранится общее количество заказов и самый крупный заказ за последний месяц. Приведем определение такой таблицы:

DESC mtd_orders; Name	Null?	Type
TOT_ORDERS	NOT NULL	NUMBER(7)
TOT_SALE_PRICE	NOT NULL	NUMBER(11,2)
MAX_SALE_PRICE	NOT NULL	NUMBER(9,2)
EUROPE_TOT_ORDERS	NOT NULL	NUMBER(7)
EUROPE_TOT_SALE_PRICE	NOT NULL	NUMBER(11,2)
EUROPE_MAX_SALE_PRICE	NOT NULL	NUMBER(9,2)
NORTHAMERICA_TOT_ORDERS	NOT NULL	NUMBER(7)
NORTHAMERICA_TOT_SALE_PRICE	NOT NULL	NUMBER(11,2)
NORTHAMERICA_MAX_SALE_PRICE	NOT NULL	NUMBER(9,2)

Каждую ночь таблица обновляется за счет информации о заказах за прошедший день. Большая часть столбцов будет изменяться каждую ночь, но столбец самого крупного заказа `max_sale_price` изменится лишь в том случае, если величина одного из заказов текущего дня превысит максимальное значение. Следующий PL/SQL-блок показывает, как можно это реализовать с помощью процедурного языка:

```
DECLARE
    tot_ord NUMBER;
    tot_price NUMBER;
    max_price NUMBER;
    prev_max_price NUMBER;
BEGIN
    SELECT COUNT(*), SUM(sale_price), MAX(sale_price)
    INTO tot_ord, tot_price, max_price
    FROM cust_order
    WHERE cancelled_dt IS NULL
       AND order_dt >= TRUNC(SYSDATE);

    UPDATE mtd_orders
    SET tot_orders = tot_orders + tot_ord,
        tot_sale_price = tot_sale_price + tot_price
    RETURNING max_sale_price INTO prev_max_price;

    IF max_price > prev_max_price THEN
        UPDATE mtd_orders
        SET max_sale_price = max_price;
    END IF;
END;
```



После того как вычислены общее количество заказов, суммарная стоимость заказов и максимальная стоимость заказа за день, обновляются столбцы `tot_orders` и `tot_sale_price` таблицы `mtd_orders`. После завершения обновления максимальная стоимость заказа из таблицы `mtd_orders` возвращается для сравнения с сегодняшней максимальной стоимостью заказа. И если последняя превышает значение, хранимое в таблице `mtd_orders`, то выполняется второй оператор `UPDATE`, обновляющий соответствующее поле.

Однако, используя `DECODE` или `CASE`, можно обновлять столбцы `tot_orders` и `tot_sale_price` и, при необходимости, обновлять столбец `max_sale_price` в одном операторе `UPDATE`. Кроме того, так как теперь у нас всего один оператор `UPDATE`, можно обобщать данные таблицы `cust_order` в подзапросе и не пользоваться `PL/SQL`:

```
UPDATE mtd_orders mtdo
SET (mtdo.tot_orders, mtdo.tot_sale_price, mtdo.max_sale_price) =
  (SELECT mtdo.tot_orders + day_tot.tot_orders,
    mtdo.tot_sale_price + NVL(day_tot.tot_sale_price, 0),
    DECODE(GREATEST(mtdo.max_sale_price,
      NVL(day_tot.max_sale_price, 0)), mtdo.max_sale_price,
      mtdo.max_sale_price, day_tot.max_sale_price)
  FROM
    (SELECT COUNT(*) tot_orders, SUM(sale_price) tot_sale_price,
      MAX(sale_price) max_sale_price
     FROM cust_order
     WHERE cancelled_dt IS NULL
       AND order_dt >= TRUNC(SYSDATE)) day_tot);
```

В данном операторе столбцу `max_sale_price` присваивается извлеченное из него значение до тех пор, пока значение, возвращаемое подзапросом, не превысит текущее значение столбца, когда же условие будет выполнено, столбцу будет присвоено значение, возвращенное подзапросом. Переформулируем запрос, используя выражение `CASE`:

```
UPDATE mtd_orders mtdo
SET (mtdo.tot_orders, mtdo.tot_sale_price, mtdo.max_sale_price) =
  (SELECT mtdo.tot_orders + day_tot.tot_orders,
    mtdo.tot_sale_price + day_tot.tot_sale_price,
    CASE WHEN day_tot.max_sale_price > mtdo.max_sale_price
      THEN day_tot.max_sale_price
      ELSE mtdo.max_sale_price END
  FROM
    (SELECT COUNT(*) tot_orders, SUM(sale_price) tot_sale_price,
      MAX(sale_price) max_sale_price
     FROM cust_order
     WHERE cancelled_dt IS NULL
       AND order_dt >= TRUNC(SYSDATE)) day_tot);
```

Используя такой подход, необходимо помнить, что присваивание столбцу извлеченного из него значения воспринимается базой данных как



изменение и может быть занесено в журнал аудита, инициировать изменение столбца `last_modified_date` и т. д.

## Выборочное обобщение

Чтобы расширить пример предыдущего раздела, давайте договоримся хранить еще и общий объем продаж для таких регионов, как Европа и Северная Америка. Изменим соответствующим образом таблицу `mtd_orders`. Добавим в нее три столбца для европейских продаж и три столбца для североамериканских продаж.

Name	Null?	Type
TOT_ORDERS	NOT NULL	NUMBER(7)
TOT_SALE_PRICE	NOT NULL	NUMBER(11,2)
MAX_SALE_PRICE	NOT NULL	NUMBER(9,2)
EUROPE_TOT_ORDERS	NOT NULL	NUMBER(7)
EUROPE_TOT_SALE_PRICE	NOT NULL	NUMBER(11,2)
EUROPE_MAX_SALE_PRICE	NOT NULL	NUMBER(9,2)
NORTHAMERICA_TOT_ORDERS	NOT NULL	NUMBER(7)
NORTHAMERICA_TOT_SALE_PRICE	NOT NULL	NUMBER(11,2)
NORTHAMERICA_MAX_SALE_PRICE	NOT NULL	NUMBER(9,2)

Что касается новых столбцов, каждый из заказов будет затрагивать или одно, или второе множество столбцов, но не оба сразу. Заказ ведь поступит или от европейского, или же от североамериканского клиента. Чтобы заполнить новые столбцы, можно написать еще два оператора `UPDATE`, по одному для региона:

```
/* Europe buckets */
UPDATE mtd_orders mtdo
SET (mtdo.europe_tot_orders, mtdo.europe_tot_sale_price,
    mtdo.europe_max_sale_price) =
(SELECT mtdo.europe_tot_orders + eur_day_tot.tot_orders,
    mtdo.europe_tot_sale_price + nvl(eur_day_tot.tot_sale_price, 0),
    CASE WHEN eur_day_tot.max_sale_price > mtdo.europe_max_sale_price
        THEN eur_day_tot.max_sale_price
        ELSE mtdo.europe_max_sale_price END
FROM
    (SELECT COUNT(*) tot_orders, SUM(co.sale_price) tot_sale_price,
        MAX(co.sale_price) max_sale_price
    FROM cust_order co, customer c
    WHERE co.cancelled_dt IS NULL
        AND co.order_dt >= TRUNC(SYSDATE)
        AND co.cust_nbr = c.cust_nbr
        AND c.region_id IN
            (SELECT region_id FROM region
             START WITH name = 'Europe'
             CONNECT BY PRIOR region_id = super_region_id)) eur_day_tot);

/* North America buckets */
UPDATE mtd_orders mtdo
```

```

SET (mtdo.northamerica_tot_orders, mtdo.northamerica_tot_sale_price,
    mtdo.northamerica_max_sale_price) =
(SELECT mtdo.northamerica_tot_orders + na_day_tot.tot_orders,
    mtdo.northamerica_tot_sale_price + nvl(na_day_tot.tot_sale_price, 0),
    CASE WHEN na_day_tot.max_sale_price > mtdo.northamerica_max_sale_price
        THEN na_day_tot.max_sale_price
        ELSE mtdo.northamerica_max_sale_price END
FROM
(SELECT COUNT(*) tot_orders, SUM(co.sale_price) tot_sale_price,
    MAX(co.sale_price) max_sale_price
FROM cust_order co, customer c
WHERE co.cancelled_dt IS NULL
    AND co.order_dt >= TRUNC(SYSDATE) - 60
    AND co.cust_nbr = c.cust_nbr
    AND c.region_id IN
        (SELECT region_id FROM region
        START WITH name = 'North America'
        CONNECT BY PRIOR region_id = super_region_id)) na_day_tot);

```

Но почему бы не сэкономить просмотр таблицы `cust_order` и не рассчитать европейские и североамериканские итоги продаж одновременно? Используем в обобщающих функциях условную логику, сделав так, чтобы на каждое из суммирований влияли только соответствующие строки. Идея подобна той, что обсуждалась в разделе «Выборочное выполнение функции», — данные обобщаются выборочно на основе информации, хранящейся в таблице:

```

UPDATE mtd_orders mtdo
SET (mtdo.northamerica_tot_orders, mtdo.northamerica_tot_sale_price,
    mtdo.northamerica_max_sale_price, mtdo.europe_tot_orders,
    mtdo.europe_tot_sale_price, mtdo.europe_max_sale_price) =
(SELECT mtdo.northamerica_tot_orders + nvl(day_tot.na_tot_orders, 0),
    mtdo.northamerica_tot_sale_price + nvl(day_tot.na_tot_sale_price, 0),
    CASE WHEN day_tot.na_max_sale_price > mtdo.northamerica_max_sale_price
        THEN day_tot.na_max_sale_price
        ELSE mtdo.northamerica_max_sale_price END,
    mtdo.europe_tot_orders + nvl(day_tot.eur_tot_orders, 0),
    mtdo.europe_tot_sale_price + nvl(day_tot.eur_tot_sale_price, 0),
    CASE WHEN day_tot.eur_max_sale_price > mtdo.europe_max_sale_price
        THEN day_tot.eur_max_sale_price
        ELSE mtdo.europe_max_sale_price END
FROM
(SELECT SUM(CASE WHEN na_regions.region_id IS NOT NULL THEN 1
    ELSE 0 END) na_tot_orders,
    SUM(CASE WHEN na_regions.region_id IS NOT NULL THEN co.sale_price
    ELSE 0 END) na_tot_sale_price,
    MAX(CASE WHEN na_regions.region_id IS NOT NULL THEN co.sale_price
    ELSE 0 END) na_max_sale_price,
    SUM(CASE WHEN eur_regions.region_id IS NOT NULL THEN 1
    ELSE 0 END) eur_tot_orders,
    SUM(CASE WHEN eur_regions.region_id IS NOT NULL THEN co.sale_price
    ELSE 0 END) eur_tot_sale_price,

```



```

MAX(CASE WHEN eur_regions.region_id IS NOT NULL THEN co.sale_price
      ELSE 0 END) eur_max_sale_price
FROM cust_order co, customer c,
(SELECT region_id FROM region
 START WITH name = 'North America'
 CONNECT BY PRIOR region_id = super_region_id) na_regions,
(SELECT region_id FROM region
 START WITH name = 'Europe'
 CONNECT BY PRIOR region_id = super_region_id) eur_regions
WHERE co.cancelled_dt IS NULL
   AND co.order_dt >= TRUNC(SYSDATE)
   AND co.cust_nbr = c.cust_nbr
   AND c.region_id = na_regions.region_id (+)
   AND c.region_id = eur_regions.region_id (+)) day_tot);

```

Это достаточно сложный оператор, так что давайте разобьем его на части. Внутри встроеного представления `day_tot` выполняется объединение таблиц `cust_order` и `customer`, а затем внешнее объединение по столбцу `customer.region_id` с каждым из двух встроенных представлений (`na_regions` и `eur_regions`), которые осуществляют иерархические запросы к таблице регионов. Таким образом, заказы европейских клиентов будут иметь отличные от `NULL` значения в столбце `eur_regions.region_id`, так как внешнее объединение найдет соответствующую строку во встроеном представлении `eur_regions`. Проводится шесть операций обобщения, три проверки для объединения со встроеным представлением `na_regions` (североамериканские заказы) и три проверки для объединения со встроеным представлением `eur_regions` (европейские заказы). Затем шесть обобщенных значений изменяют шесть столбцов таблицы `mtd_orders`.

Данный оператор можно (и нужно) объединить с оператором из предыдущего примера (обновляющего три первых столбца), чтобы создать оператор `UPDATE`, который воздействовал бы на все столбцы таблицы `mtd_orders` за один проход таблицы `cust_order`. Для приложений хранения данных, в которых каждую ночь за короткий период времени должны обрабатываться большие объемы данных, применение такого подхода может означать успех вместо провала.

## Ошибки деления на ноль

Общей нормой при написании кода является элегантная обработка неправильных значений данных. Одной из наиболее часто встречающихся арифметических ошибок является `ORA-01476`: делитель равен нулю. Извлекается ли значение из столбца, передается через связанную переменную или же возвращается вызовом функции, в любом случае необходимо заключать делитель в оболочку — `DECODE` или `CASE`, как показано в следующем примере:

```

SELECT p.part_nbr, SYSDATE + (p.inventory_qty /
   DECODE(my_pkg.get_daily_part_usage(p.part_nbr), NULL, 1,

```



```
0, 1, my_pkg.get_daily_part_usage(p.part_nbr))) anticipated_shortage_dt
FROM part p
WHERE p.inventory_qty > 0;
```

**Функция DECODE гарантирует, что делитель отличен от нуля. Приведем CASE-версию оператора:**

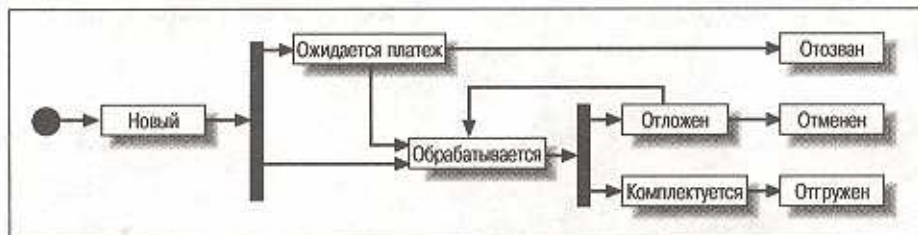
```
SELECT p.part_nbr, SYSDATE + (p.inventory_qty /
CASE WHEN my_pkg.get_daily_part_usage(p.part_nbr) > 0
THEN my_pkg.get_daily_part_usage(p.part_nbr)
ELSE 1 END) anticipated_shortage_dt
FROM part p
WHERE p.inventory_qty > 0;
```

Если вас беспокоит, что функция `get_daily_part_usage` повторно вызывается для каждой детали, для которой выполнено условие, поместите вызов функции во встроенное представление:

```
SELECT parts.part_nbr, SYSDATE + (parts.inventory_qty /
CASE WHEN parts.daily_part_usage > 0
THEN parts.daily_part_usage
ELSE 1 END) anticipated_shortage_dt
FROM
(SELECT p.part_nbr part_nbr, p.inventory_qty inventory_qty,
my_pkg.get_daily_part_usage(p.part_nbr) daily_part_usage
FROM part p
WHERE p.inventory_qty > 0) parts;
```

## Переходы из одного состояния в другое

В некоторых случаях ограничение налагается не только на то, какие значения могут использоваться для столбца, но и на порядок их изменения. Рассмотрим схему, на которой приведены разрешенные изменения состояния заказа (рис. 9.1).



**Рис. 9.1.** Допустимые изменения состояния заказа

Как видите, из состояния «Обрабатывается» заказ может перейти только в состояние «Отложен» или «Комплектуется». Вместо того чтобы в каждом приложении для изменения состояния заказа применять условную логику, давайте напишем пользовательскую функцию, которая будет возвращать соответствующее состояние в зависимости от

текущего состояния и типа перехода. Для данного примера определяем два типа перехода: положительный (positive – POS) и отрицательный (negative – NEG). Например, из состояния «Отложен» заказ может совершить положительный переход к состоянию «Обрабатывается» или же отрицательный к «Отменен». Если заказ находится в одном из итоговых состояний (Отозван, Отменен, Отгружен), возвращается текущее состояние. Приведем DECODE-версию функции PL/SQL:

```

FUNCTION get_next_order_state(ord_nbr in NUMBER,
    trans_type in VARCHAR2 DEFAULT 'POS')
RETURN VARCHAR2 is
    next_state VARCHAR2(20) := 'UNKNOWN';
BEGIN
    SELECT DECODE(status,
        'REJECTED', status,
        'CANCELLED', status,
        'SHIPPED', status,
        'NEW', DECODE(trans_type, 'NEG', 'AWAIT_PAYMENT', 'PROCESSING'),
        'AWAIT_PAYMENT', DECODE(trans_type, 'NEG', 'REJECTED', 'PROCESSING'),
        'PROCESSING', DECODE(trans_type, 'NEG', 'DELAYED', 'FILLED'),
        'DELAYED', DECODE(trans_type, 'NEG', 'CANCELLED', 'PROCESSING'),
        'FILLED', DECODE(trans_type, 'POS', 'SHIPPED', 'UNKNOWN'),
        'UNKNOWN')
    INTO next_state
    FROM cust_order
    WHERE order_nbr = ord_nbr;

    RETURN next_state;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN next_state;
END get_next_order_state;
```

Для того чтобы иметь возможность воспользоваться CASE-версией функции, вы должны работать с Oracle9i, так как грамматика языка PL/SQL предыдущих версий не включает в себя выражение CASE:

```

FUNCTION get_next_order_state(ord_nbr in NUMBER,
    trans_type in VARCHAR2 DEFAULT 'POS')
RETURN VARCHAR2 is
    next_state VARCHAR2(20) := 'UNKNOWN';
BEGIN
    SELECT CASE
        WHEN status = 'REJECTED' THEN status
        WHEN status = 'CANCELLED' THEN status
        WHEN status = 'SHIPPED' THEN status
        WHEN status = 'NEW' AND trans_type = 'NEG' THEN 'AWAIT_PAYMENT'
        WHEN status = 'NEW' AND trans_type = 'POS' THEN 'PROCESSING'
        WHEN status = 'AWAIT_PAYMENT' AND trans_type = 'NEG' THEN 'REJECTED'
        WHEN status = 'AWAIT_PAYMENT' AND trans_type = 'POS' THEN 'PROCESSING'
        WHEN status = 'PROCESSING' AND trans_type = 'NEG' THEN 'DELAYED'
        WHEN status = 'PROCESSING' AND trans_type = 'POS' THEN 'FILLED'
```

```

    WHEN status = 'DELAYED' AND trans_type = 'NEG' THEN 'CANCELLED'
    WHEN status = 'DELAYED' AND trans_type = 'POS' THEN 'PROCESSING'
    WHEN status = 'FILLED' AND trans_type = 'POS' THEN 'SHIPPED'
    ELSE 'UNKNOWN'
END
INTO next_state
FROM cust_order
WHERE order_nbr = ord_nbr;

RETURN next_state;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN next_state;
END get_next_order_state;

```

Данный пример обрабатывает только простой случай, когда существует всего два пути перехода из каждого состояния, но он иллюстрирует подход к управлению переходом данных из одного состояния в другое. Предыдущую функцию можно, например, применить для изменения состояния заказа в базе данных (посредством оператора UPDATE) в случае его успешного перехода в другое состояние:

```

UPDATE cust_order
SET status = my_pkg.get_next_order_state(order_nbr, 'POS')
WHERE order_nbr = 1107;

```



# 10

## Разделы, объекты и коллекции

В Oracle8 был введен ряд новых возможностей для работы с большими базами данных и объектно-реляционными конструкциями. В Oracle8i и Oracle9i эти функциональные возможности расширены и усовершенствованы. В этой главе мы поговорим о декомпозиции, необходимой при реализации больших баз данных, и об объектах и коллекциях, которые облегчают хранение и передачу сложных типов данных.

### Разделение таблиц

За последние 15 лет размеры жестких дисков с 10 мегабайт возросли до 100 гигабайт и продолжают увеличиваться. Размеры дисковых массивов быстро приближаются к отметке в 100 терабайт. Но неважно, насколько большое хранилище вам доступно, потребности всегда будут превышать возможности. При увеличении объема базы данных становится все сложнее выполнять каждодневные операции. Например, выделение времени и ресурсов на пересоздание индекса, содержащего 100 миллионов записей, может представлять серьезную проблему. До выхода Oracle8 администраторы баз данных вручную разбивали большие таблицы на несколько более мелких. Хотя в запросе эти части могли скрываться под специальным представлением (называемым *распределенным представлением (partition view)*), все операторы DML должны были выполняться для отдельных таблиц, так что схема разделения была видна разработчикам и пользователям базы данных.

Начиная с версии Oracle8 предоставляются средства разделения таблицы на несколько частей при сохранении видимости единой таблицы. Каждый элемент такой таблицы называется *разделом (partition)*, и, хотя все разделы должны совместно использовать столбцы, ограни-

чения, индексы и триггеры, каждый раздел может иметь свои уникальные параметры хранения. Администраторы, выделяя место для хранения и выполняя резервное копирование, обычно имеют дело с отдельными разделами, а вот разработчики могут выбрать: работать ли им с целой таблицей или же с отдельными разделами.

## Базовые сведения о разделах

Разработчики и администраторы баз данных занимались разделением таблиц задолго до того, как появился Oracle8. Обычно разделение таблиц в рамках одной базы данных производится для повышения производительности и упрощения администрирования, в то время как разделение таблиц между базами данных облегчает распространение данных. Например, данные о продажах могут быть разделены по регионам, и каждый раздел может храниться в базе данных, расположенной в соответствующем региональном офисе. Центральное хранилище данных может собирать данные о продажах всех офисов для составления отчетов и принятия решения, но вполне разумно, чтобы рабочие данные были распределены по нескольким локальным базам.

Разделение таблицы на множества строк, как в примере с данными о продажах, где местоположение данных определяется по значению столбца «отдел сбыта», называется *горизонтальным разделением (horizontal partitioning)*. Можно производить разбиение и на множества столбцов, называемое *вертикальным разделением (vertical partitioning)*. Например, такие конфиденциальные данные, как размер заработной платы и номер социального страхования, могут быть отделены от таблицы служащих в специальную таблицу с ограниченным доступом. При выполнении вертикального разделения необходимо включать в каждое множество столбцы первичного ключа. Поэтому в отличие от горизонтального разделения, при котором разделы содержат непересекающиеся подмножества данных, вертикальное разделение требует дублирования определенных данных в каждом разделе.

Вертикальное и горизонтальное разделение таблиц внутри базы данных Oracle и между несколькими базами данных может быть осуществлено вручную. Команда, введенная в Oracle8, имеет дело только с горизонтальным разделением в рамках одной базы данных.

## Разделение таблиц

При разделении таблица превращается из физического объекта в виртуальное понятие. На самом деле это уже больше не таблица, а набор разделов. Но поскольку все части таблицы должны иметь одинаковые атрибуты и определения ограничений, можно работать с набором разделов так, как если бы речь шла об одной таблице. Единственное, чем могут отличаться разделы, — это параметры хранения, такие как размер экстенда и размещение табличного пространства. Благодаря этим различиям можно реализовывать такие интересные сценарии хране-



ния, как хранение редко используемых разделов на CD, в то время как разделы, к которым обращаются часто, хранятся на диске. Кроме того, можно воспользоваться сегментированием буферного кэша, сохраняя наиболее активные разделы в буфере, чтобы они всегда были в памяти, тогда как остальные разделы могут выгружаться из буфера. Также можно сделать отдельные разделы временно недоступными, никак не затрагивая доступность остальных разделов, что значительно увеличивает гибкость.

В зависимости от выбранной схемы разделения один или несколько столбцов таблицы должны быть назначены *ключом раздела (partition key)*. Значения ключа раздела указывают, к какому разделу относится данная строка. Oracle также использует информацию ключа раздела совместно с инструкцией WHERE для определения того, какие разделы просматривать при выполнении операторов SELECT, UPDATE и DELETE (подробная информация приведена ниже в разделе «Отсечение разделов» данной главы).

## Разделение индексов

Что же происходит с индексами при разделении таблиц? Выбор за вами: можно оставить любой из индексов нетронутым (такие индексы будем называть *глобальными (global index)*) или же разбить его на части, соответствующие разделам таблицы (*локальные индексы (local-index)*). Что касается глобальных индексов, можно разбить их не так, как была разбита таблица. Если вы примете во внимание тот факт, что как древовидные (b-tree), так и растровые (bit-map) индексы могут использоваться одновременно, то поймете, что все может очень усложниться. Если к разделенной таблице применяется оператор SELECT, UPDATE или DELETE, то для поиска строк оптимизатор может воспользоваться различными способами:

1. Использовать глобальный индекс (если таковой существует и его столбцы упомянуты в операторе SQL) для нахождения требуемых строк в одном или нескольких разделах.
2. Просматривать локальный индекс для каждого раздела для определения того, содержит ли раздел искомые строки.
3. Определить подмножество разделов, которые могут содержать искомые строки, а затем обратиться к локальным индексам этих разделов.

Хотя простейшим решением кажется использование глобального индекса, это может быть проблематично. Так как глобальные индексы распространяются на все разделы таблицы, на них неблагоприятно влияет текущая работа с разделами. Например, если раздел разбивается на несколько частей или два раздела сливаются в один, все глобальные индексы таблицы помечаются как непригодные для использования и, прежде чем их можно будет снова использовать, они должны



быть перестроены. Особенно неприятно, когда глобальные индексы применяются по умолчанию для первичных ключей разделенной таблицы. Используйте вместо глобальных индексов локальные. Для поддержания целостности разделенных таблиц можно применять механизм уникальных локальных индексов.<sup>1</sup>

## Методы разделения

Для того чтобы выполнить горизонтальное разделение таблицы (или индекса), необходимо задать набор правил, по которым Oracle будет определять, к какому разделу относится каждая из строк. В следующем разделе рассматриваются четыре типа разделения, доступные в Oracle9i.

### Разделение по диапазону

Первая схема разделения таблиц, введенная в Oracle8 и называемая *разделением по диапазону* (*range partitioning*), позволяет разбить таблицу по диапазонам значений одного или нескольких столбцов таблицы. Простейший и наиболее широко распространенный способ разделения по диапазону – это разбиение по столбцу дат. Рассмотрим следующий оператор DDL:

```
CREATE TABLE cust_order (
  order_nbr NUMBER(7) NOT NULL,
  cust_nbr NUMBER(5) NOT NULL,
  order_dt DATE NOT NULL,
  sales_emp_id NUMBER(5) NOT NULL,
  sale_price NUMBER(9,2),
  expected_ship_dt DATE,
  cancelled_dt DATE,
  ship_dt DATE,
  status VARCHAR2(20))
)
PARTITION BY RANGE (order_dt)
(PARTITION orders_1999
  VALUES LESS THAN (TO_DATE('01-JAN-2000', 'DD-MON-YYYY'))
  TABLESPACE ord1,
 PARTITION orders_2000
  VALUES LESS THAN (TO_DATE('01-JAN-2001', 'DD-MON-YYYY'))
  TABLESPACE ord2,
 PARTITION orders_2001
  VALUES LESS THAN (TO_DATE('01-JAN-2002', 'DD-MON-YYYY'))
  TABLESPACE ord3);
```

При использовании такой схемы разбиения все заказы, сделанные до 2000 года, попадут в раздел `orders_1999`; заказы 2000 года будут храниться в разделе `orders_2000`, а заказы 2001 года – в разделе `orders_2001`.

---

<sup>1</sup> При создании первичного ключа можно указать существующий индекс, вместо того чтобы заставлять Oracle создавать новый глобальный индекс.

## Хеш-разделение

Иногда требуется разбить большую таблицу, но в ней нет столбца, по которому можно было бы провести разделение по диапазону. Появившееся в Oracle8i *хеш-разделение* (*hash partitioning*) позволяет указать только количество разделов и столбцы, по которым проводится разделение (ключ разделения), а собственно распределением строк по разделам занимается Oracle. При вставке строк в разделенную таблицу Oracle пытается равномерно распределить данные между разделами, применяя к данным ключа разделения хеш-функцию. Значение, возвращаемое хеш-функцией, определяет в какой раздел попадет строка. Если столбцы ключа разделения включены в инструкцию WHERE оператора SELECT, DELETE или UPDATE, то для определения того, какой раздел просматривать, Oracle использует хеш-функцию. Покажем, как можно разделить таблицу деталей хешированием столбца `part_nbr`:

```
CREATE TABLE part (  
  part_nbr VARCHAR2(20) NOT NULL,  
  name VARCHAR2(50) NOT NULL,  
  supplier_id NUMBER(6) NOT NULL,  
  inventory_qty NUMBER(6) NOT NULL,  
  status VARCHAR2(10) NOT NULL,  
  inventory_qty NUMBER(6),  
  unit_cost NUMBER(8,2)  
  resupply_date DATE)  
PARTITION BY HASH (part_nbr)  
(PARTITION part1 TABLESPACE p1,  
 PARTITION part2 TABLESPACE p2,  
 PARTITION part3 TABLESPACE p3,  
 PARTITION part4 TABLESPACE p4);
```

Чтобы равномерно распределить данные между разделами, необходимо выбирать в качестве ключей разделения столбцы большой мощности (*high cardinality*). Говорят, что множество столбцов обладает большой мощностью, если количество различных значений достаточно велико по сравнению с размером таблицы.<sup>1</sup> Выбор в качестве ключа разделения столбца большой мощности гарантирует равномерное распределение данных по разделам. В противном случае разделы получаются несбалансированными, что делает производительность непредсказуемой и усложняет администрирование.

## Композитное разделение

Если вы разрываетесь между разделением по диапазону и хеш-разделением, можно использовать понемногу и от одного и от другого. По-

---

<sup>1</sup> Самой высокой мощностью обладает уникальный ключ, так как в нем все строки имеют различные значения. Примером столбца низкой мощности является столбец стран в таблице клиентов, содержащей миллионы записей.



явившееся в Oracle8i *композиционное разделение* (*composite partitioning*) позволяет создавать несколько диапазоновых разделов, каждый из которых содержит два или более хеш-подраздела. Композиционное разделение удобно использовать тогда, когда тип данных таблицы позволяет выполнить разделение по диапазону, но вам хотелось бы большей детализации, чем это возможно при применении чисто диапазонного разделения. Например, разумно разбить таблицу заказов по годам. Если вы хотите создать несколько разделов для каждого года, можно разбить каждый год на подразделы, хешированием разделив номера клиентов на четыре части. Следующий пример уточняет выполненное ранее разделение по диапазону, образуя подразделы на основе хеш-функции номера клиента:

```
CREATE TABLE cust_order (  
  order_nbr NUMBER(7) NOT NULL,  
  cust_nbr NUMBER(5) NOT NULL,  
  order_dt DATE NOT NULL,  
  sales_emp_id NUMBER(5) NOT NULL,  
  sale_price NUMBER(9,2),  
  expected_ship_dt DATE,  
  cancelled_dt DATE,  
  ship_dt DATE,  
  status VARCHAR2(20))  
PARTITION BY RANGE (order_dt)  
SUBPARTITION BY HASH (cust_nbr) SUBPARTITIONS 4  
STORE IN (order_sub1, order_sub2, order_sub3, order_sub4)  
(PARTITION orders_1999  
  VALUES LESS THAN (TO_DATE('01-JAN-2000', 'DD-MON-YYYY'))  
  (SUBPARTITION orders_1999_s1 TABLESPACE order_sub1,  
   SUBPARTITION orders_1999_s2 TABLESPACE order_sub2,  
   SUBPARTITION orders_1999_s3 TABLESPACE order_sub3,  
   SUBPARTITION orders_1999_s4 TABLESPACE order_sub4),  
 PARTITION orders_2000  
  VALUES LESS THAN (TO_DATE('01-JAN-2001', 'DD-MON-YYYY'))  
  (SUBPARTITION orders_2000_s1 TABLESPACE order_sub1,  
   SUBPARTITION orders_2000_s2 TABLESPACE order_sub2,  
   SUBPARTITION orders_2000_s3 TABLESPACE order_sub3,  
   SUBPARTITION orders_2000_s4 TABLESPACE order_sub4),  
 PARTITION orders_2001  
  VALUES LESS THAN (TO_DATE('01-JAN-2002', 'DD-MON-YYYY'))  
  (SUBPARTITION orders_2001_s1 TABLESPACE order_sub1,  
   SUBPARTITION orders_2001_s2 TABLESPACE order_sub2,  
   SUBPARTITION orders_2001_s3 TABLESPACE order_sub3,  
   SUBPARTITION orders_2001_s4 TABLESPACE order_sub4));
```

При использовании композиционного разделения все данные физически хранятся в подразделах, а разделы, как и сами таблицы, становятся виртуальными.



## Разделение по списку

Введенное в Oracle9i *разделение по списку (list partitioning)* позволяет разбить таблицу по различным значениям определенного столбца. Например, таблица, хранящая итоговые данные о продажах по товарам, штатам и месяцам/годам, может быть разделена по географическим регионам:

```
CREATE TABLE sales_fact (  
  state_cd VARCHAR2(3) NOT NULL,  
  month_cd NUMBER(2) NOT NULL,  
  year_cd NUMBER(4) NOT NULL,  
  product_cd VARCHAR2(10) NOT NULL,  
  tot_sales NUMBER(9,2) NOT NULL)  
PARTITION BY LIST (state_cd)  
(PARTITION sales_newengland VALUES ('CT', 'RI', 'MA', 'NH', 'ME', 'VT')  
  TABLESPACE s1,  
  PARTITION sales_northwest VALUES ('OR', 'WA', 'MT', 'ID', 'WY', 'AK')  
  TABLESPACE s2,  
  PARTITION sales_southwest VALUES ('NV', 'UT', 'AZ', 'NM', 'CO', 'HI')  
  TABLESPACE s3,  
  PARTITION sales_southeast VALUES ('FL', 'GA', 'AL', 'SC', 'NC', 'TN', 'WV')  
  TABLESPACE s4,  
  PARTITION sales_east VALUES ('PA', 'NY', 'NJ', 'MD', 'DE', 'VA', 'KY', 'OH')  
  TABLESPACE s5,  
  PARTITION sales_california VALUES ('CA')  
  TABLESPACE s6,  
  PARTITION sales_south VALUES ('TX', 'OK', 'LA', 'AR', 'MS')  
  TABLESPACE s7,  
  PARTITION sales_midwest VALUES ('ND', 'SD', 'NE', 'KS', 'MN', 'WI', 'IA',  
  'IL', 'IN', 'MI', 'MO')  
  TABLESPACE s8);
```

Разделение по списку удобно для данных низкой мощности, когда количество различных значений столбца мало по сравнению с количеством строк. В отличие от разделения по диапазону и хеш-разделения, при которых ключ разделения может состоять из нескольких столбцов, ключ разделения по списку всегда ограничен одним столбцом. Казалось бы, композитное разделение могло бы предлагать на первом уровне как разделение по диапазону, так и разделение по списку, но пока в Oracle реализовано только совместное использование разделения по диапазону и хеш-разделения.

## Определение разделов

При написании операторов SQL для разделенных таблиц можно рассматривать разделы как единую виртуальную таблицу или же указывать в операторах названия конкретных разделов. Если оператор DML применяется к виртуальной таблице, оптимизатор Oracle определяет раздел или разделы, с которыми будет производиться работа. Для опе-

ратора INSERT при определении того, в какой раздел помещать какую строку, оптимизатор использует значения ключа разделения. В операторах UPDATE, DELETE и SELECT для определения разделов, которые будут просматриваться, оптимизатор использует условия инструкции WHERE, а также информацию локальных и глобальных индексов.

Если вы знаете, что оператор DML затрагивает только один раздел, и знаете название этого раздела, то можете использовать инструкцию PARTITION, чтобы сообщить оптимизатору, с каким разделом работать. Например, если вы хотите обобщить заказы за 2000 год и знаете, что для таблицы cust\_order выполнено разделение по диапазону по годам, то можете написать такой запрос:

```
SELECT COUNT(*) tot_orders, SUM(sale_price) tot_sales
FROM cust_order PARTITION (orders_2000)
WHERE cancelled_dt IS NULL;
```

Обратите внимание, что в инструкции WHERE запроса не указан диапазон дат, хотя таблица и содержит данные за несколько лет. Так как указан раздел orders\_2000, запрос будет обобщать только данные за 2000 год, и нет необходимости проверять дату каждого заказа.

Если для таблицы выполнено композитное разделение, можно использовать инструкцию SUBPARTITION для указания отдельного подраздела таблицы. Например, следующий оператор удаляет все строки подраздела orders\_2000\_s1 таблицы cust\_order, для которой было выполнено композитное разделение:

```
DELETE FROM cust_order SUBPARTITION (orders_2000_s1);
```

Можно также использовать инструкцию PARTITION для удаления всего набора подразделов, входящих в данный раздел:

```
DELETE FROM cust_order PARTITION (orders_2000);
```

Такой оператор удалит все строки подразделов orders\_2000\_s1, orders\_2000\_s2, orders\_2000\_s3 и orders\_2000\_s4 таблицы cust\_order.

Приведем несколько фактов, о которых необходимо помнить при работе с разделенными таблицами:

- Если оптимизатор определяет, что для вычисления инструкции WHERE оператора UPDATE, DELETE или SELECT необходим просмотр двух или более разделов, разделы таблицы и/или индекса могут просматриваться параллельно. Поэтому в зависимости от доступных Oracle системных ресурсов просмотр разделенной таблицы может быть выполнен гораздо быстрее, чем просмотр неразделенной таблицы.
- Так как хеш-разделение распределяет данные по разделам случайным образом,<sup>1</sup> непонятно, как использовать инструкцию PARTITION

<sup>1</sup> В действительности данные распределяются не случайным образом. Но вам будет казаться именно так, поскольку у вас нет доступа к хеш-функции.



для таблиц с хеш-разделением и инструкцию `SUBPARTITION` для таблиц с композитным разделением – ведь неизвестно, куда какие данные попали. Единственный разумный план действий, приходящий в голову, относится к случаю, когда необходимо изменить все строки таблицы, но размер сегмента отката недостаточен для изменения всех строк в одной транзакции. Тогда можно выполнить оператор `UPDATE` или `DELETE` для каждого раздела или подраздела, а после завершения работы каждого оператора выполнить инструкцию `COMMIT`.

- Администратор базы данных в любой момент может объединить несколько разделов, разбить раздел на подразделы или удалить раздел. Поэтому будьте внимательны при явном указании названий разделов в операторах DML. В противном случае оператор может не выполниться или, что еще хуже, выполниться не для того набора данных, из-за того что вы не знали об объединении или разбиении разделов. Стоит совместно с вашим администратором БД выработать правила по использованию названий разделов в операторах DML.

Если вы хотите обратиться к одному разделу или подразделу, но не хотите засорять код названиями разделов, подумайте о создании представлений, скрывающих названия разделов:

```
CREATE VIEW cust_order_2000 AS
SELECT *
FROM cust_order PARTITION (orders_2000);
```

Затем можно применять операторы SQL к представлению:

```
SELECT order_nbr, cust_nbr, sale_price, order_dt
FROM cust_order_2000
WHERE quantity > 100;
```

## Отсечение разделов

Даже если вы не указываете в операторе SQL конкретный раздел, сам факт разделения таблицы может влиять на организацию обращения к ней. Если оператор SQL затрагивает одну или несколько разделенных таблиц, оптимизатор Oracle пытается использовать при выполнении оператора информацию инструкции `WHERE` для исключения из рассмотрения некоторых разделов. Этот процесс, называемый *отсечением разделов* (*partition pruning*),<sup>1</sup> ускоряет выполнение запроса за счет игнорирования разделов, не соответствующих условиям инструкции `WHERE`. Оптимизатор использует сведения из определения таблицы в сочетании с информацией об условиях инструкции `WHERE`. Например, имея такое определение таблицы:

```
CREATE TABLE tab1 (
  col1 NUMBER(5) NOT NULL,
```

---

<sup>1</sup> А также *исключением разделов* (*partition elimination*).



```
col2 DATE NOT NULL,  
col3 VARCHAR2(10) NOT NULL)  
PARTITION BY RANGE (col2)  
(PARTITION tab1_1998  
VALUES LESS THAN (TO_DATE('01-JAN-1999', 'DD-MON-YYYY'))  
TABLESPACE t1,  
PARTITION tab1_1999  
VALUES LESS THAN (TO_DATE('01-JAN-2000', 'DD-MON-YYYY'))  
TABLESPACE t1,  
PARTITION tab1_2000  
VALUES LESS THAN (TO_DATE('01-JAN-2001', 'DD-MON-YYYY'))  
TABLESPACE t3,  
PARTITION tab1_2001  
VALUES LESS THAN (TO_DATE('01-JAN-2002', 'DD-MON-YYYY'))  
TABLESPACE t4);
```

и такой запрос:

```
SELECT col1, col2, col3  
FROM tab1  
WHERE col2 > TO_DATE('01-OCT-2000', 'DD-MON-YYYY');
```

оптимизатор исключит из рассмотрения разделы tab1\_1998 и tab1\_1999, так как ни один из этих разделов не может содержать строки, для которых значение столбца col2 превышает 1 октября 2000 года.

Чтобы оптимизатор мог принимать подобные решения, инструкция WHERE должна ссылаться хотя бы на один столбец из множества столбцов, составляющих ключ разделения. Хотя это может показаться очень простым условием, на самом деле далеко не все запросы к разделенной таблице будут содержать ключ разделения. Например, если существует уникальный индекс для столбца col1 таблицы tab1 из предыдущего примера, то наиболее эффективный доступ к данным обеспечивал бы следующий запрос:

```
SELECT col1, col2, col3  
FROM tab1  
WHERE col1 = 1578;
```

Если же индекс для столбца col1 был определен как локальный, то Oracle будет вынужден просмотреть локальный индекс каждого раздела, чтобы найти тот, который хранит значение 1578. Если у вас также есть информация о ключе разделения (в данном случае – col2), можно включить его в запрос, чтобы оптимизатор имел возможность отсеять ненужные разделы:

```
SELECT col1, col2, col3  
FROM tab1  
WHERE col1 = 1578  
AND col2 > TO_DATE('01-JAN-2001', 'DD-MON-YYYY');
```

За счет дополнительного условия оптимизатор исключает из рассмотрения разделы `tab1_1998`, `tab1_1999` и `tab1_2000`. Oracle будет просматривать только один уникальный индекс для раздела `tab1_2001`, вместо того чтобы просматривать уникальные индексы всех четырех разделов таблицы. Конечно, для этого необходимо знать, что дата, относящаяся к значению 1578, имеет значение `col2`, превышающее 1 января 2001 года. Если есть возможность предоставить достоверную дополнительную информацию о ключах разделения, следует ее использовать, в противном случае – просто доверьтесь оптимизатору. Выполнив инструкцию `EXPLAIN PLAN` для оператора DML, работающего с разделенной таблицей, можно посмотреть, какие разделы были выбраны оптимизатором.

Чтобы посмотреть, какие именно разделы рассматриваются оптимизатором, необходимо после инструкции `EXPLAIN PLAN` выполнить запрос к таблице `plan_table`, указав несколько столбцов, являющихся характеристиками разделов. Исследуем следующий запрос к таблице `tab1`:

```
EXPLAIN PLAN
SET STATEMENT_ID = 'qry1' FOR
SELECT col1, col2, col3
FROM tab1
WHERE col2 BETWEEN TO_DATE('01-JUL-1999', 'DD-MON-YYYY')
AND TO_DATE('01-JUL-2000', 'DD-MON-YYYY');
```

При обращении к таблице `plan_table` включаем столбцы `partition_start` и `partition_end` всюду, где поле операции начинается с 'PARTITION':

```
SELECT lpad(' ', 2 * level) || operation || ' ' ||
options || ' ' || object_name ||
DECODE(SUBSTR(operation, 1, 9), 'PARTITION',
' FROM ' || partition_start ||
' TO ' || partition_stop, ' ') "exec plan"
FROM plan_table
CONNECT BY PRIOR id = parent_id
START WITH id = 0 AND statement_id = 'qry1';
```

```
exec plan
```

```
-----
SELECT STATEMENT
PARTITION RANGE ITERATOR FROM 2 TO 3
TABLE ACCESS FULL TAB1
```

Значение `PARTITION RANGE` в столбце `operation` вместе со значением `ITERATOR` в столбце `options` означают, что в план выполнения будет включено несколько разделов.<sup>1</sup> Значения столбцов `partition_start` и `partition_end` (2 и 3, соответственно) показывают, что оптимизатор исключил разде-

<sup>1</sup> Если оптимизатор исключил из рассмотрения все разделы, кроме одного, столбец `options` будет содержать значение 'SINGLE'. Если ни один из разделов не отсечен, то `options` содержит значение 'ALL'.



лы 1 и 4 (tab1\_1998 и tab1\_2001).<sup>1</sup> Учитывая то, что инструкция WHERE указывает диапазон дат от 1 июля 1999 года до 1 июля 2000 года, можно сказать, что оптимизатор правильно отсекает все разделы, которые не могут участвовать в получении результата.

## Объекты и коллекции

В версии Oracle8 к чисто реляционному серверу баз данных добавлены объектно-ориентированные возможности. *Объектные типы (object types)* и *коллекции (collections)* появились в Oracle8, а затем были значительно усовершенствованы в Oracle8i и Oracle9i, так что сегодня их можно считать полнофункциональными.<sup>2</sup> Теперь Oracle рассматривает свой процессор баз данных как объектно-реляционный, то есть база данных может сочетать в себе как реляционные конструкции, такие как таблицы и ограничения, так и объектно-ориентированные конструкции, такие как объектные типы, коллекции и ссылки.

### Объектные типы

Объектный тип – это определенный пользователем тип данных, комбинирующий данные и соответствующие методы для создания сложных сущностей. В этом объектные типы похожи на определения классов в объектно-ориентированном языке, таком как C++ или Java. Однако в отличие от Java и C++, объектные типы Oracle имеют встроенный механизм сохранения: можно определить таблицу для хранения объектного типа в базе данных. Соответственно, объектные типы Oracle могут управляться непосредственно через SQL.

Удобнее всего продемонстрировать синтаксис и свойства объектного типа на примере. Следующий оператор DDL создает объектный тип, используемый для моделирования ценных бумаг:

```
CREATE TYPE equity AS OBJECT (  
    issuer_id NUMBER,  
    ticker VARCHAR2(6),  
    outstanding_shares NUMBER,  
    last_close_price NUMBER(9,2),  
    MEMBER PROCEDURE  
    apply_split(split_ratio IN VARCHAR2));
```

<sup>1</sup> Числа в столбцах partition\_start и partition\_end соответствуют числам в столбце partition\_position таблицы user\_tab\_partitions, поэтому можно обратиться к этой таблице, чтобы определить названия разделов, включенных в план выполнения.

<sup>2</sup> Например, в Oracle8.0 объектные типы не поддерживали наследование, а коллекции не могли быть вложенными (например, массив массивов), из-за чего эти первые шаги Oracle к объектной ориентации были восприняты достаточно холодно.



Объектный тип `equity` имеет четыре свойства и один метод (процедуру). Тело процедуры `apply_split` определено в операторе `CREATE TYPE BODY`. Следующий пример показывает, как можно определить процедуру `apply_split`:

```
CREATE TYPE BODY equity AS
  MEMBER PROCEDURE apply_split(split_ratio IN VARCHAR2) IS
    from_val NUMBER;
    to_val NUMBER;
  BEGIN
    /* разделение значения дробления акций на компоненты */
    to_val := SUBSTR(split_ratio, 1, INSTR(split_ratio, ':') - 1);
    from_val := SUBSTR(split_ratio, INSTR(split_ratio, ':') + 1);

    /* применение изменений для выпущенных в обращение акций */
    SELF.outstanding_shares :=
      (SELF.outstanding_shares * to_val) / from_val;

    /* применение изменений для цены на момент закрытия биржи */
    SELF.last_close_price :=
      (SELF.last_close_price * from_val) / to_val;
  END apply_split;
END;
```

Ключевое слово `SELF` используется для идентификации текущего экземпляра объектного типа `equity`. Хотя это и необязательно, рекомендуем использовать его в коде, чтобы было понятно, что вы изменяете или ссылаетесь на данные текущего экземпляра. Чуть позже мы подробнее поговорим о том, как вызывать методы.

Экземпляры типа `equity` создаются при помощи конструктора по умолчанию, который имеет то же имя, что и объектный тип, и предполагает один параметр для каждого атрибута объектного типа. В `Oracle9i` нет возможности формировать собственные конструкторы для объектных типов. Следующий блок `PL/SQL` показывает, как конструктор по умолчанию создает экземпляр объектного типа `equity`:

```
DECLARE
  eq equity := NULL;
BEGIN
  eq := equity(198, 'ACMW', 1000000, 13.97);
END;
```

Конструкторы объектных типов можно вызывать из операторов `DML`. Следующий пример обращается к таблице `issuer` для нахождения эмитента с названием 'ACME Wholesalers' и использует извлеченное значение `issuer_id` для создания экземпляра типа `equity`:

```
DECLARE
  eq equity := NULL;
BEGIN
  SELECT equity(i.issuer_id, 'ACMW', 1000000, 13.97)
```

```

    INTO eq
  FROM issuer i
  WHERE i.name = 'ACME Wholesalers';
END;
```

В трех последующих разделах кратко описываются некоторые способы включения объектных типов в базу данных и/или ее логику.

## Объекты в качестве атрибутов

Объектный тип может использоваться наряду со встроенными типами данных Oracle в качестве атрибута таблицы. В определении таблицы `common_stock` объектный тип `equity` использован как атрибут:

```

CREATE TABLE common_stock (
  security_id NUMBER NOT NULL,
  security equity NOT NULL,
  issue_date DATE NOT NULL,
  currency_cd VARCHAR2(5) NOT NULL);
```

При добавлении записей в такую таблицу необходимо использовать конструктор объектного типа, как показано в следующем операторе `INSERT`:

```

INSERT INTO common_stock (security_id, security, issue_date, currency_cd)
VALUES (1078, equity(198, 'ACMW', 1000000, 13.97), SYSDATE, 'USD');
```

Чтобы посмотреть на атрибуты объекта `equity`, необходимо создать псевдоним таблицы и указать ссылку на псевдоним, название объектного атрибута таблицы и атрибут объектного типа. Напишем запрос, извлекающий из таблицы `common_stock` значение атрибута `security_id`, относящегося к таблице `common_stock`, и атрибута `ticker`, относящегося к объекту `equity`:

```

SELECT c.security_id security_id,
       c.security.ticker ticker
FROM common_stock c;

SECURITY_ID TICKER
-----
1078 ACMW
```

## Объектные таблицы

Можно не только встраивать объектные типы в таблицы наряду с другими атрибутами, но и построить таблицу специально для хранения экземпляров объектного типа. Такие таблицы называются *объектными* (*object table*) и создаются посредством указания ссылки на объектный тип в операторе `CREATE TABLE` с использованием ключевого слова `OF`:

```

CREATE TABLE equities OF equity;
```

Таблицу `equities` можно заполнить данными с помощью конструктора объектного типа `equity`. Для заполнения таблицы также можно использовать существующие экземпляры объектного типа `equity`. Например, заполним таблицу `equities`, используя столбец `security` таблицы `common_stock`:

```
INSERT INTO equities
SELECT c.security FROM common_stock c;
```

При обращении к таблице `equities` можно непосредственно ссылаться на атрибуты объектного типа, как если бы это была обычная таблица:

```
SELECT issuer_id, ticker
FROM equities;

ISSUER_ID TICKER
-----
198 ACMW
```

Если вы хотите извлечь из таблицы `equities` данные в виде экземпляра объекта `equity`, а не набора атрибутов, используйте функцию `VALUE`, возвращающую объект. Следующий PL/SQL-блок извлекает из таблицы `equities` объект, для которого значение атрибута `ticker` равно `'ACMW'`:

```
DECLARE
    eq equity := NULL;
BEGIN
    SELECT VALUE(e)
    INTO eq
    FROM equities e
    WHERE ticker = 'ACMW';
END;
```

То есть объектные таблицы вобрали в себя лучшее с обеих сторон: их можно рассматривать и как реляционные таблицы, и как набор объектов.



Заметьте, что функция `VALUE` использует псевдоним таблицы. Применение `VALUE` требует использования псевдонимов таблиц.

Теперь, когда у нас в базе данных хранятся объекты, давайте поговорим о том, как вызвать определенный ранее метод `apply_split`. Прежде чем вызывать метод, необходимо найти соответствующий объект в таблице и сообщить объекту, что он должен запустить свою процедуру `apply_split`. Напишем на основе предыдущего примера PL/SQL-блок, который находит в таблице `equities` объект, для которого значение `ticker` равно `'ACMW'`, вызывает его метод `apply_split` и записывает его обратно в таблицу:

```
DECLARE
    eq equity := NULL;
```



```
BEGIN
  SELECT VALUE(e)
  INTO eq
  FROM equities e
  WHERE ticker = 'ACMW';

  /* дробление акций 2:1 */
  eq.apply_split('2:1');

  /* сохранение измененного объекта */
  UPDATE equities e
  SET e = eq
  WHERE ticker = 'ACMW';
END;
```

Важно понимать, что процедура `apply_split` не работает непосредственно с данными таблицы `equities`; она работает с копией объекта, хранящейся в памяти. После того как процедура `apply_split` выполнена для копии, оператор `UPDATE` записывает данные поверх объекта таблицы `equities`, на который указывает локальная переменная `eq`, тем самым сохраняя измененную версию объекта.

## Объектные параметры

Вне зависимости от того, хотите ли вы хранить объектные типы в базе данных на постоянной основе или нет, вы можете использовать их как носители для передачи данных внутри приложения или между приложениями. Объектные типы могут применяться как входные параметры и возвращаемые типы хранимых процедур и функций PL/SQL. Кроме того, операторы `SELECT` могут обрабатывать и возвращать объектные типы, даже если ни одна из таблиц в инструкции `FROM` не содержит объектных типов. Поэтому объектные типы могут применяться для того, чтобы создать видимость объектно-ориентированного дизайна для чисто реляционной модели базы данных.

Чтобы показать, как это может работать, давайте создадим для нашей тестовой базы данных API (application programming interface – интерфейс программирования приложений), который может принимать и возвращать объектные типы для нахождения и создания заказов клиентов. Начнем с определения необходимых объектных типов:

```
CREATE TYPE customer_obj AS OBJECT
(cust_nbr NUMBER,
 name VARCHAR2(30));

CREATE TYPE employee_obj AS OBJECT
(emp_id NUMBER,
 name VARCHAR2(50));

CREATE TYPE order_obj AS OBJECT
(order_nbr NUMBER,
 customer customer_obj,
```

```

salesperson employee_obj,
order_dt DATE,
price NUMBER,
status VARCHAR2(20));

CREATE TYPE line_item_obj AS OBJECT (
    part_nbr VARCHAR2(20),
    quantity NUMBER(8,2));

```

**Используя эти определения объектов, создадим пакет PL/SQL, содержащий процедуры и функции, обеспечивающие жизненный цикл заказа клиента:**

```

CREATE PACKAGE order_lifecycle AS
    FUNCTION create_order(v_cust_nbr IN NUMBER, v_emp_id IN NUMBER)
        RETURN order_obj;
    PROCEDURE cancel_order(v_order_nbr IN NUMBER);
    FUNCTION get_order(v_order_nbr IN NUMBER) RETURN order_obj;
    PROCEDURE add_line_item(v_order_nbr IN NUMBER,
        v_line_item IN line_item_obj);
END order_lifecycle;

CREATE PACKAGE BODY order_lifecycle AS
    FUNCTION create_order(v_cust_nbr IN NUMBER, v_emp_id IN NUMBER)
        RETURN order_obj IS
        ord_nbr NUMBER;
    BEGIN
        SELECT seq_order_nbr.NEXTVAL INTO ord_nbr FROM DUAL;

        INSERT INTO cust_order (order_nbr, cust_nbr, sales_emp_id,
            order_dt, expected_ship_dt, status)
        SELECT ord_nbr, c.cust_nbr, e.emp_id,
            SYSDATE, SYSDATE + 7, 'NEW'
        FROM customer c, employee e
        WHERE c.cust_nbr = v_cust_nbr
            AND e.emp_id = v_emp_id;

        RETURN order_lifecycle.get_order(ord_nbr);
    END create_order;

    PROCEDURE cancel_order(v_order_nbr IN NUMBER) IS
    BEGIN
        UPDATE cust_order SET cancelled_dt = SYSDATE,
            expected_ship_dt = NULL, status = 'CANCELED'
        WHERE order_nbr = v_order_nbr;
    END cancel_order;

    FUNCTION get_order(v_order_nbr IN NUMBER) RETURN order_obj IS
        ord order_obj := NULL;
    BEGIN
        SELECT order_obj(co.order_nbr,
            customer_obj(c.cust_nbr, c.name),
            employee_obj(e.emp_id, e.fname || ' ' || e.lname),
            co.order_dt, co.sale_price, co.status)

```

```
INTO ord
FROM cust_order co, customer c, employee e
WHERE co.order_nbr = v_order_nbr
      AND co.cust_nbr = c.cust_nbr
      AND co.sales_emp_id = e.emp_id;

RETURN ord;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RETURN ord;
END get_order;

PROCEDURE add_line_item(v_order_nbr IN NUMBER,
  v_line_item IN line_item_obj) IS
BEGIN
  INSERT INTO line_item (order_nbr, part_nbr, qty)
  VALUES (v_order_nbr, v_line_item.part_nbr,
    v_line_item.quantity);
END add_line_item;
END order_lifecycle;
```

С точки зрения пользователя API в базе данных хранятся объекты, несмотря на то что сама база данных – чисто реляционная. Если вы с отращением думаете о возможности использования в схеме вашей базы данных объектных типов, предложенный подход может быть хорошей альтернативой обращению к Java-программистам для непосредственного управления реляционными данными.

## Типы коллекций

В процессе создания традиционного реляционного проекта отношения «один ко многим», такие как подразделение, включающее множество сотрудников, или заказ, состоящий из нескольких пунктов, реализуются как отдельные таблицы, при этом дочерняя таблица содержит внешний ключ для родительской. В схеме тестовой БД каждая строка таблицы *line\_item* «знает», к какому заказу она относится, благодаря внешнему ключу, но строка таблицы *cust\_order* не имеет прямых сведений о пунктах заказов. Начиная с Oracle8 подобные отношения могут быть включены в родительскую таблицу посредством *коллекций* (*collection*). В версиях Oracle8 и выше доступны два типа коллекций: *переменные массивы* (*variable arrays*), использующиеся для упорядоченных ограниченных наборов данных, и *вложенные таблицы* (*nested tables*), применяемые для неупорядоченных и неограниченных наборов данных.

### Переменные массивы

Переменные массивы (часто называемые *varrays*) – это массивы, хранящиеся в таблице. Все элементы переменного массива должны быть одного типа, они ограничены максимальным размером, и доступ к ним



осуществляется позиционно. Переменные массивы могут содержать или стандартный тип данных Oracle, такой как DATE или VARCHAR2, или определенный пользователем объектный тип. Создадим переменный массив и используем его для столбца таблицы:

```
CREATE TYPE resupply_dates AS VARRAY(100) OF DATE;

CREATE TABLE part_c (
  part_nbr VARCHAR2(20) NOT NULL,
  name VARCHAR2(50) NOT NULL,
  supplier_id NUMBER(6),
  unit_cost NUMBER(8,2),
  inventory_qty NUMBER(6),
  restocks resupply_dates);
```

Наряду с описательной информацией о детали каждая строка таблицы part\_c может хранить до 100 дат, соответствующих получению товара от поставщиков.

## Вложенные таблицы

Как и для переменных массивов, элементы вложенных таблиц должны быть одного типа. Однако в отличие от переменных массивов, вложенные таблицы не имеют максимального размера, и доступ к ним осуществляется не позиционно. Строки вложенной таблицы должны содержать только один столбец, который может быть определен как стандартный тип данных или как объектный тип. Если использован объектный тип, получается, что вложенная таблица как бы содержит несколько столбцов, так как объектный тип может содержать несколько атрибутов. Определим тип вложенной таблицы, содержащей объектный тип:

```
CREATE TYPE line_item_obj AS OBJECT (
  part_nbr VARCHAR2(20),
  quantity NUMBER(8,2));

CREATE TYPE line_item_tbl AS TABLE OF line_item_obj;
```

Мы создали вложенную таблицу для объектов line\_item, и теперь можно встроить ее в таблицу cust\_order:

```
CREATE TABLE cust_order_c (
  order_nbr NUMBER(8) NOT NULL,
  cust_nbr NUMBER(6) NOT NULL,
  sales_emp_id NUMBER(6) NOT NULL,
  order_dt DATE NOT NULL,
  sale_price NUMBER(9,2),
  order_items line_item_tbl)
NESTED TABLE order_items STORE AS order_items_table;
```

Используя вложенную таблицу, вы включили пункты заказов в таблицу cust\_order, и в таблице line\_item больше нет необходимости. Далее

вы узнаете, что Oracle предоставляет возможность в случае необходимости отсоединить коллекцию `order_items`.

## Создание коллекции

Хотя определения таблиц предыдущего раздела выглядят достаточно просто, не очень понятно, как заполнять получившиеся таблицы данными. Каждый раз, когда вы хотите создать экземпляр коллекции, необходимо использовать ее конструктор – системную функцию, имеющую то же название, что и коллекция. Конструктор принимает один или несколько элементов, причем для переменных массивов количество элементов не может превышать максимальный размер массива. Например, добавление строки в таблицу `part_c`, содержащую столбец – переменный массив, можно произвести следующим образом:

```
INSERT INTO part_c (part_nbr, name, supplier_id, unit_cost,
    inventory_qty, restocks)
VALUES ('GX5-2786-A2', 'Spacely Sprocket', 157, 75, 22,
    resupply_dates(TO_DATE('03-SEP-1999', 'DD-MON-YYYY'),
        TO_DATE('22-APR-2000', 'DD-MON-YYYY'),
        TO_DATE('21-MAR-2001', 'DD-MON-YYYY')));
```

Конструктор `resupply_dates` вызывается с тремя параметрами, по одному для каждой даты получения партии деталей. Если вы используете для формирования запросов распознающее коллекции инструментальное средство, например Oracle SQL\*Plus, то можете обращаться к коллекции напрямую, и результаты будут отформатированы утилитой:

```
SELECT part_nbr, restocks
FROM part_c
WHERE name = 'Spacely Sprocket';
```

PART_NBR	RESTOCKS
-----	
GX5-2786-A2	RESUPPLY_DATES('03-SEP-99', '22-APR-00', '21-MAR-01')

Обработка вложенных таблиц производится аналогично. Приведем пример вставки новой строки в таблицу `cust_order_c`, содержащую столбец – вложенную таблицу:

```
INSERT INTO cust_order_c (order_nbr, cust_nbr, sales_emp_id,
    order_dt, sale_price, order_items)
VALUES (1000, 9568, 275,
    TO_DATE('21-MAR-2001', 'DD-MON-YYYY'), 15753,
    line_item_tbl(
        line_item_obj('A675-015', 25),
        line_item_obj('GX5-2786-A2', 1),
        line_item_obj('X378-9JT-2', 3)));
```

Если вы посмотрите внимательно, то заметите, что в действительности вызываются два разных конструктора: один создает вложенную таб-



лицу `line_item_tbl`, а второй создает каждый из трех экземпляров объектного типа `line_item_obj`. Не забывайте, что вложенная таблица — это таблица объектов `line_item_obj`. Конечным результатом является единственная строка `cust_order_c`, содержащая коллекцию из трех пунктов заказа.

## Разборка коллекций

Даже если ваша группа разработчиков виртуозно работает с коллекциями в рамках вашей базы данных, было бы неразумно предполагать, что все различные приложения и средства, обращающиеся к вашим данным (программы загрузки и извлечения данных, программы составления отчетов, специальные утилиты для формирования запросов и т. д.), будут их корректно обрабатывать. Применяя методику, называемую *разборкой коллекции* (*collection unnesting*), можно представить содержимое коллекции в виде строк обычной таблицы. Используя выражение `TABLE`, напомним запрос, который выделяет вложенную таблицу `order_items` из таблицы `cust_order_c`:

```
SELECT co.order_nbr, co.cust_nbr, co.order_dt, li.part_nbr, li.quantity
FROM cust_order_c co,
     TABLE(co.order_items) li;
```

ORDER_NBR	CUST_NBR	ORDER_DT	PART_NBR	QUANTITY
1000	9568	21-MAR-01	A675-015	25
1000	9568	21-MAR-01	GX5-2786-A2	1
1000	9568	21-MAR-01	X378-8JT-2	3

Заметьте, что в явном объединении двух наборов данных нет необходимости, так как члены коллекции уже сопоставлены строке таблицы `cust_order_c`.

Чтобы сделать выделенное множество данных доступным пользователям, можно поместить предыдущий запрос в представление:

```
CREATE VIEW cust_order_line_items AS
SELECT co.order_nbr, co.cust_nbr, co.order_dt, li.part_nbr, li.quantity
FROM cust_order_c co,
     TABLE(co.order_items) li;
```

Теперь пользователи могут взаимодействовать с вложенной таблицей через представление, используя стандартный SQL, например:

```
SELECT *
FROM cust_order_line_items
WHERE part_nbr like 'X%';
```

ORDER_NBR	CUST_NBR	ORDER_DT	PART_NBR	QUANTITY
1000	9568	21-MAR-01	X378-8JT-2	3



## Обращение к коллекциям

Вы уже знаете, как поместить данные в коллекцию, но как извлечь их оттуда? Oracle предоставляет специально для этого выражение `TABLE`.<sup>1</sup> Выражение `TABLE` может использоваться в инструкциях `FROM`, `WHERE` и `HAVING`, обеспечивая возможность ссылки на столбец, являющийся вложенной таблицей или переменным массивом, так, как если бы это была отдельная таблица. Следующий запрос извлекает даты повторных поставок (из столбца `restocks`), которые только что были добавлены в таблицу `part_c`:

```
SELECT *
FROM TABLE(SELECT restocks
              FROM part_c
              WHERE part_nbr = 'GX5-2786-A2');

COLUMN_VALUE
-----
03-SEP-1999
22-APR-2000
21-MAR-2001
```

Чтобы лучше понять функцию выражения `TABLE`, извлечем переменный массив `restocks` непосредственно из таблицы `part_c`:

```
SELECT restocks
FROM part_c
WHERE part_nbr = 'GX5-2786-A2'

RESTOCKS
-----
RESUPPLY_DATES('03-SEP-99', '22-APR-00', '21-MAR-01')
```

Как видите, результирующее множество состоит из одной строки, содержащей массив дат, в то время как предыдущий запрос выделяет переменный массив и представляет каждый его элемент как строку с одним столбцом.

Так как переменный массив содержит встроенный тип данных, а не объектный тип, необходимо дать ему название, чтобы на него можно было явно ссылаться в операторах SQL. В таких случаях Oracle по умолчанию присваивает содержимому переменного массива псевдоним `'column_value'`. В следующем примере мы используем этот псевдоним.

Пусть необходимо найти все детали, поставленные в определенный день. Используя выражение `TABLE`, выполним связанный подзапрос к переменному массиву `restocks`, чтобы проверить, входит ли в него нужная дата:

---

<sup>1</sup> В версиях, предшествующих 8i, выражение `TABLE` называлось `THE`. В данной книге будет использоваться только `TABLE`.

```

SELECT p1.part_nbr, p1.name
FROM part_c p1
WHERE TO_DATE('03-SEP-1999', 'DD-MON-YYYY') IN
  (SELECT column_value FROM TABLE(SELECT restocks FROM part_c p2
    WHERE p2.part_nbr = p1.part_nbr));

```

PART_NBR	NAME
-----	-----
GX5-2786-A2	Spacely Sprocket

## Изменение коллекций

При желании изменить содержимое коллекции можно пойти двумя путями: заменить всю коллекцию или же только ее отдельные элементы. Если коллекция – это переменный массив, можно заменить только всю коллекцию целиком. Для этого извлеките содержимое массива, измените данные и обновите таблицу. Приведем оператор, изменяющий даты пополнения запасов для детали с номером 'GX5-2786-A2'. Обратите внимание, что переменный массив пересоздается полностью:

```

UPDATE part_c
SET restocks = resupply_dates(TO_DATE('03-SEP-1999', 'DD-MON-YYYY'),
  TO_DATE('25-APR-2000', 'DD-MON-YYYY'),
  TO_DATE('21-MAR-2001', 'DD-MON-YYYY'))
WHERE part_nbr = 'GX5-2786-A2';

```

Если же речь идет о вложенной таблице, то можно выполнить оператор DML для отдельных элементов коллекции. Например, следующий оператор добавляет дополнительную строку заказа во вложенную таблицу `cust_order_c` для заказа под номером 1000:

```

INSERT INTO TABLE(SELECT order_items FROM cust_order_c
  WHERE order_nbr = 1000)
VALUES (line_item_obj('T25-ASM', 1));

```

Если требуется изменить данные во вложенной таблице, используйте выражение `TABLE` для создания множества данных, состоящего из номеров деталей заказа под номером 1000, а затем измените элемент с заданным номером детали:

```

UPDATE TABLE(SELECT order_items FROM cust_order_c
  WHERE order_nbr = 1000) oi
SET oi.quantity = 2
WHERE oi.part_nbr = 'T25-ASM';

```

Аналогично можно использовать то же самое множество данных для удаления элементов из коллекции:

```

DELETE FROM TABLE(SELECT order_items FROM cust_order_c
  WHERE order_nbr = 1000) oi
WHERE oi.part_nbr = 'T25-ASM';

```

# 11

## PL/SQL

Существует множество хороших книг, во всех подробностях описывающих язык PL/SQL.<sup>1</sup> Так как наша книга посвящена Oracle SQL, сосредоточимся на использовании PL/SQL в операторах SQL и применении SQL в программах PL/SQL.

### Что такое PL/SQL?

PL/SQL – это процедурный язык программирования корпорации Oracle, который объединяет следующие элементы:

- Логические конструкции, такие как IF-THEN-ELSE и WHILE.
- Операторы DML, встроенные функции и инструкции.
- Операторы управления транзакциями, такие как COMMIT и ROLLBACK.
- Операторы управления курсором.
- Операторы манипулирования объектами и коллекциями.

Появившись в версии 6.0 в качестве языка сценариев, PL/SQL стал неотъемлемой частью сервера Oracle 7.0 (PL/SQL версии 2.0). Так как версия 7.0 поддерживает возможность компиляции и хранения программ PL/SQL на сервере, Oracle начинает использовать язык для обеспечения функциональности сервера и содействия в установке и конфигурировании базы данных. При включении PL/SQL версии 2.1 в редакцию сервера Oracle 7.1 добавляется новая возможность, особенно по-

---

<sup>1</sup> Например, «Oracle PL/SQL Programming» Стивена Феерштайна (Steven Feuerstein), издательство O'Reilly.



лезная для SQL-программистов, – вызов хранимых функций PL/SQL из операторов SQL (далее об этом будет рассказано подробно).

Наряду с поддержкой новых возможностей в каждой новой версии PL/SQL Oracle начинает поставлять готовые пакеты PL/SQL, что позволяет программистам браться за решение более сложных задач и способствует интеграции с разнообразными продуктами Oracle. Такие совокупности хранимых процедур и функций, имеющие название *Oracle Supplied Packages*, позволяют (среди прочего):

- Взаимодействовать и администрировать Oracle Advanced Queuing.
- Составлять расписание периодического выполнения задач базы данных.
- Манипулировать большими объектами Oracle (LOB).
- Читать внешние файлы и записывать в них.
- Взаимодействовать с функциями репликации Oracle (Oracle Advanced Replication).
- Генерировать динамические операторы SQL.
- Генерировать и разбирать XML-файлы.
- Генерировать команды LDAP.

Постоянно расширяющийся набор функциональных возможностей языка PL/SQL вкупе с большим количеством поставляемых пакетов породил мощную программную среду для баз данных. Если вы составляете отчеты, пишете сценарии загрузки данных или заказные приложения, в вашем проекте наверняка есть место для PL/SQL.

## Процедуры, функции и пакеты

Хотя PL/SQL можно использовать для написания сценариев, называемых также *анонимными блоками* (*anonymous blocks*), темой этой главы являются подпрограммы PL/SQL, хранящиеся на сервере Oracle. Подпрограммы PL/SQL, хранящиеся в базе данных, могут быть двух типов: *хранимые процедуры* (*stored procedures*) или *хранимые функции* (*stored functions*).<sup>1</sup> Хранимые функции и процедуры, по существу, идентичны, разница лишь в следующем:

- Хранимые функции имеют возвращаемый тип, а процедуры – нет.
- Так как хранимые функции возвращают значение, их можно использовать в выражениях, а процедуры – нет.

Хранимые процедуры и функции могут компилироваться индивидуально, а могут группироваться в *пакеты* (*packages*). Пакеты – это не просто удобный способ объединения взаимосвязанных функциональных возможностей, они важны по следующим причинам:

---

<sup>1</sup> Триггеры базы данных тоже относятся к хранимому PL/SQL, но они выходят за рамки повествования данной книги.

- Пакеты загружаются в память целиком, что увеличивает вероятность того, что процедура или функция будет находиться в памяти при вызове.
- Пакеты могут содержать закрытые элементы, позволяя скрыть логику.
- Помещение процедур и функций в пакеты избавляет от необходимости перекомпиляции всех функций и процедур, ссылающихся на вновь перекомпилированную.
- Внутри пакетов названия процедур и функций могут быть перегружены, в то время как автономные процедуры и функции не допускают перегрузки.
- Внутри пакетов функции и процедуры могут быть проверены на побочные эффекты в момент компиляции, а не во время выполнения, что повышает производительность.

Если все вышеперечисленное не убедило вас поместить хранимые процедуры и функции в пакеты, скажем вам еще кое-что на основе нашего опыта работы с PL/SQL начиная с версии 2.0: вы никогда не пожалеете, что сохранили код PL/SQL внутри пакетов, но рано или поздно пожалеете, если не сделаете этого.

Пакеты состоят из двух частей: *спецификации пакета* (*package specification*), определяющей заголовки открытых процедур и функций пакета, и *тела пакета* (*package body*), содержащего код открытых процедур и функций; кроме того, пакет может содержать код закрытых функций и процедур, не включенных в спецификацию пакета. Приведем для наглядности простой пример спецификации пакета:

```
CREATE OR REPLACE PACKAGE my_pkg AS
  PROCEDURE my_proc(arg1 IN VARCHAR2);

  FUNCTION my_func(arg1 IN NUMBER) RETURN VARCHAR2;
END my_pkg;
```

**и соответствующего тела пакета:**

```
CREATE OR REPLACE PACKAGE BODY my_pkg AS
  FUNCTION my_private_func(arg1 IN NUMBER) RETURN VARCHAR2 IS
    return_val VARCHAR2(20);
  BEGIN
    SELECT col1 INTO return_val
    FROM tab2
    WHERE col2 = arg1;

    RETURN return_val;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RETURN 'NOT FOUND';
  END my_private_func;

  PROCEDURE my_proc(arg1 IN VARCHAR2) IS
```



```
BEGIN
  UPDATE tab1 SET col1 = col1 + 1
  WHERE col2 = arg1;
END my_proc;

FUNCTION my_func(arg1 IN NUMBER) RETURN VARCHAR2 IS
BEGIN
  RETURN my_private_func(arg1);
END my_func;
END my_pkg;
```

Как видите, пакет `my_pkg` содержит одну открытую процедуру и одну открытую функцию. Спецификация пакета включает в себя названия и типы параметров процедуры и функции, а также тип возвращаемого функцией значения, но не содержит никакого кода. Тело пакета включает в себя логику реализации открытой функции и открытой процедуры, а также закрытую функцию (`my_private_func`), которая доступна только внутри тела пакета.

## Вызов хранимых функций из запросов

Как уже говорилось, хранимые функции могут вызываться из операторов SQL. Так как хранимые функции могут, в свою очередь, вызывать хранимые процедуры, можно сказать, что и хранимые процедуры, хотя и не напрямую, но могут вызываться из операторов SQL. Хранимые функции могут использоваться в выражениях, поэтому они могут быть включены в запрос всюду, где разрешены выражения, в том числе в:

- Инструкцию `SELECT`
- Инструкцию `WHERE`
- Инструкции `GROUP BY` и `HAVING`
- Инструкцию `ORDER BY`
- Инструкцию `START WITH` (для иерархических запросов)
- Инструкцию `FROM` (опосредованно, используя встроенные представления или операторы `TABLE`)

Одним из наиболее часто встречающихся применений хранимых функций является выделение часто используемой функциональности для облегчения повторного применения кода и упрощения сопровождения. Представьте, например, что вы работаете с большой командой над построением многоуровневого приложения. Чтобы упростить интеграцию уровней, было решено, что все даты будут передаваться в виде количества миллисекунд, прошедших с 1 января 1970 года. Можно включить логику преобразования во все запросы, например:

```
SELECT co.order_nbr, co.cust_nbr, co.sale_price,
       ROUND((co.order_dt - TO_DATE('01011970', 'MMDDYYYY')) * 86400 * 1000)
FROM cust_order co
WHERE ship_dt = TRUNC(SYSDATE);
```



Но это достаточно утомительно и может вызвать проблемы в случае возможного последующего изменения логики. Можно создать пакет и включить в него функции по переводу значений из внутреннего формата дат Oracle в нужный формат:

```
CREATE OR REPLACE PACKAGE BODY pkg_util AS
  FUNCTION translate_date(dt IN DATE) RETURN NUMBER IS
  BEGIN
    RETURN ROUND((dt - TO_DATE('01011970', 'MMDDYYYY')) * 86400 * 1000);
  END translate_date;

  FUNCTION translate_date(dt IN NUMBER) RETURN DATE IS
  BEGIN
    RETURN TO_DATE('01011970', 'MMDDYYYY') + (dt / (86400 * 1000));
  END translate_date;
END pkg_util;
```

Заметьте, что пакет содержит две одинаково названные функции, одна из которых требует параметр типа DATE и возвращает значение типа NUMBER, в то время как вторая принимает параметр типа NUMBER, а возвращает DATE. Такая стратегия, называемая *перегрузкой* (*overloading*), является единственно возможной для функций, содержащихся в пакете.

Теперь ваша команда разработчиков может использовать эти функции каждый раз, когда необходимо преобразовать формат даты, например:

```
SELECT co.order_nbr, co.cust_nbr, co.sale_price,
       pkg_util.translate_date(co.order_dt) utc_order_dt
FROM cust_order co
WHERE co.ship_dt = TRUNC(SYSDATE);
```

Хранимые функции также часто используются для того, чтобы упростить сложную условную логику (IF-THEN-ELSE) и скрыть ее от оператора SQL. Предположим, что вам нужно составить отчет с подробным описанием заказов всех клиентов за последний месяц. Необходимо упорядочить заказы, используя: столбец ship\_dt для отправленных заказов; столбец expected\_ship\_dt для заказов, у которых дата отправки определена и не относится к прошедшему времени; текущую дату, если expected\_ship\_dt относится к прошедшему времени, или же столбец order\_dt, если дата отправки заказа еще не определена. Можно использовать оператор CASE в инструкции ORDER BY:

```
SELECT co.order_nbr, co.cust_nbr, co.sale_price
FROM cust_order co
WHERE co.order_dt > TRUNC(SYSDATE, 'MONTH')
AND co.cancelled_dt IS NULL
ORDER BY
CASE
  WHEN co.ship_dt IS NOT NULL THEN co.ship_dt
  WHEN co.expected_ship_dt IS NOT NULL
    AND co.expected_ship_dt > SYSDATE
```

```

        THEN co.expected_ship_dt
    WHEN co.expected_ship_dt IS NOT NULL
        THEN GREATEST(SYSDATE, co.expected_ship_dt)
    ELSE co.order_dt
END;
```

Однако очевидны две проблемы:

1. Получилась очень сложная инструкция ORDER BY.
2. Если вы захотите использовать эту логику где-то еще, могут возникнуть проблемы с ее сопровождением.

Давайте лучше добавим в пакет `pkg_util` хранимую функцию, которая возвращает соответствующую дату для указанного заказа:

```

FUNCTION get_best_order_date(ord_dt IN DATE, exp_ship_dt IN DATE,
    ship_dt IN DATE) RETURN DATE IS
BEGIN
    IF ship_dt IS NOT NULL THEN
        RETURN ship_dt;
    ELSIF exp_ship_dt IS NOT NULL AND exp_ship_dt > SYSDATE THEN
        RETURN exp_ship_dt;
    ELSIF exp_ship_dt IS NOT NULL THEN
        RETURN SYSDATE;
    ELSE
        RETURN ord_dt;
    END IF;
END get_best_order_date;
```

Теперь эту функцию можно вызывать и из инструкции SELECT, и из ORDER BY:

```

SELECT co.order_nbr, co.cust_nbr, co.sale_price,
    pkg_util.get_best_order_date(co.order_dt, co.expected_ship_dt,
        co.ship_dt) best_date
FROM cust_order co
WHERE co.order_dt > TRUNC(SYSDATE, 'MONTH')
    AND co.cancelled_dt IS NULL
ORDER BY pkg_util.get_best_order_date(co.order_dt, co.expected_ship_dt,
    co.ship_dt);
```

Если вас смущает то, что хранимая функция вызывается для каждой строки дважды с одними и теми же параметрами, вы всегда можете поместить извлечение данных во встроенное представление, а впоследствии отсортировать их:

```

SELECT orders.order_nbr, orders.cust_nbr,
    orders.sale_price, orders.best_date
FROM
    (SELECT co.order_nbr order_nbr, co.cust_nbr cust_nbr,
        co.sale_price sale_price,
        pkg_util.get_best_order_date(co.order_dt, co.expected_ship_dt,
```



```
co.ship_dt) best_date
FROM cust_order co
WHERE co.order_dt > TRUNC(SYSDATE, 'MONTH')
AND co.cancelled_dt IS NULL) orders
ORDER BY orders.best_date;
```

## Хранимые функции и представления

Так как представление – это не что иное, как хранимый запрос, а хранимые функции могут вызываться из инструкции `SELECT` запроса, столбцы представления могут соответствовать вызовам хранимой функции. Вы убережете сообщество пользователей от затруднений и, кроме того, получите еще одно интересное преимущество. Рассмотрим определение представления, которое содержит вызовы разных хранимых функций:

```
CREATE OR REPLACE VIEW vw_example
(col1, col2, col3, col4, col5, col6, col7, col8)
AS SELECT t1.col1,
t1.col2,
t2.col3,
t2.col4,
pkg_example.func1(t1.col1, t2.col3),
pkg_example.func2(t1.col2, t2.col4),
pkg_example.func3(t1.col1, t2.col3),
pkg_example.func4(t1.col2, t2.col4)
FROM tab1 t1, tab2 t2
WHERE t1.col1 = t2.col3;
```

Первые четыре столбца представления соответствуют столбцам таблиц `tab1` и `tab2`, значения же остальных столбцов генерируются вызовами различных функций пакета `pkg_example`. Если пользователь выполняет такой запрос:

```
SELECT col2, col4, col7
FROM vw_example
WHERE col1 = 1001;
```

то, хотя представление содержит четыре столбца, отображающих вызовы функций, в действительности выполняется только одна хранимая функция (`pkg_example.func3`). Дело в том, что когда запрос выполняется для представления, сервер Oracle создает новый запрос, комбинируя исходный запрос и определение представления. В данном случае фактически выполняется следующий запрос:

```
SELECT t1.col2,
t2.col4,
pkg_example.func3(t1.col1, t2.col3)
FROM tab1 t1, tab2 t2
WHERE t1.col1 = 1001 AND t1.col1 = t2.col3;
```



Следовательно, представление может содержать десятки вызовов хранимых функций, но выполнены будут только те, на которые явно ссылаются запросы.<sup>1</sup>

## Избегайте объединения таблиц

Представьте, что вы создали для пользователей ряд представлений для формирования отчетов и выполнения специальных запросов, и один из пользователей просит добавить в одно из представлений новый столбец. Новый столбец относится к таблице, которая еще не включена в инструкцию FROM, и нужен он только для одного отчета, выпускаемого раз в месяц. Можно добавить таблицу в инструкцию FROM, добавить столбец в инструкцию SELECT и добавить условия объединения в инструкцию WHERE. Но тогда каждый запрос, обращенный к представлению, будет содержать новую таблицу несмотря на то, что большая часть запросов не ссылается на новый столбец.

В качестве альтернативы можно предложить написание хранимой функции, которая обращается к новой таблице и возвращает нужный столбец. Эту хранимую функцию добавим в инструкцию SELECT (при этом нет никакой необходимости в добавлении новой таблицы в инструкцию FROM). Давайте несколько расширим предыдущий пример. Пусть вас интересует столбец col6 таблицы tab3, тогда можно добавить в пакет pkg\_example новую функцию:

```
FUNCTION func5(param1 IN NUMBER) RETURN VARCHAR2 IS
    ret_val VARCHAR2(20);
BEGIN
    SELECT col6 INTO ret_val
    FROM tab3
    WHERE col5 = param1;

    RETURN ret_val;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN null;
END func5;
```

Теперь добавим в представление столбец, соответствующий новой функции:

```
CREATE OR REPLACE VIEW vw_example
(col1, col2, col3, col4, col5, col6, col7, col8, col9)
AS SELECT t1.col1,
    t1.col2,
```

---

<sup>1</sup> Поэтому никогда не следует использовать при работе с представлениями инструкцию SELECT \*. Всегда явно указывайте названия необходимых столбцов, чтобы сервер не терял время на генерацию данных, которые никогда не будут использованы.

```
t2.col3,  
t2.col4,  
pkg_example.func1(t1.col1, t2.col3),  
pkg_example.func2(t1.col2, t2.col4),  
pkg_example.func3(t1.col1, t2.col3),  
pkg_example.func4(t1.col2, t2.col4),  
pkg_example.func5(t2.col3)  
FROM tab1 t1, tab2 t2  
WHERE t1.col1 = t2.col3;
```

Пользователи получили доступ к столбцу col6 таблицы tab3, при этом таблица tab3 не была добавлена в инструкцию FROM представления. Для пользователей, которые не ссылаются на новый столбец представления (col9), производительность запросов, использующих представление vw\_example, не изменится.

Несмотря на то что изначально столбец предназначался для одного отчета, не удивляйтесь, когда и другие пользователи решат включить его в свои запросы. При увеличении частоты использования столбца можно подумать о том, чтобы перестать использовать хранимую функцию и включить таблицу tab3 в инструкцию FROM. Благодаря использованию представления вы сможете выполнить эту операцию так, что пользователям не придется ничего менять в своих запросах.

## Ограничения на вызов PL/SQL из SQL

Вызов хранимых функций из SQL – это важная и мощная возможность, но необходимо понимать, что ее применение может привести к непредвиденным последствиям. Представьте, например, что один из ваших коллег написал хранимую функцию, которая, принимая номер детали, определяет, сколько раз эта деталь встречается в открытых заказах. Функция включена в пакет подпрограмм:

```
CREATE OR REPLACE PACKAGE pkg_util AS  
    FUNCTION get_part_order_qty(pno IN VARCHAR2) RETURN NUMBER;  
END pkg_util;
```

Вам поручают составить еженедельный отчет об инвентаризации, и вы хотите использовать функцию в одном из запросов:

```
SELECT p.part_nbr, p.name, s.name, p.inventory_qty,  
       pkg_util.get_part_order_qty(p.part_nbr) open_order_qty  
FROM part p, supplier s  
WHERE p.supplier_id = s.supplier_id  
ORDER BY s.name, p.part_nbr;
```

Однако, запустив запрос, вы с удивлением видите такую ошибку:

```
ORA-14551: cannot perform a DML operation inside a query
```



Проверив тело пакета, вы видите, что функция `get_part_order_qty`, наряду с вычислением количества вхождений детали в открытые заказы, генерирует заявку на пополнение запаса детали (вставляя запись в таблицу `part_order`), если вычисленное значение превышает количество деталей на складе. Если бы Oracle позволила оператору выполняться, то в результате выполнения запроса была бы без вашего на то согласия (и даже уведомления об этом) изменена информация в базе данных.

## Уровень чистоты

Чтобы определить, может ли хранимая функция, вызываемая из оператора SQL, иметь непредвиденные результаты, Oracle присваивает каждой функции *уровень чистоты* (*purity level*), который представляет собой ответы на четыре вопроса:

1. Читает ли функция данные из таблиц базы данных?
2. Ссылается ли функция на какие-то глобальные переменные пакета?
3. Пишет ли функция в какие-то таблицы базы данных?
4. Изменяет ли функция какие-то глобальные переменные пакета?

При каждом отрицательном ответе в уровень чистоты функции добавляется соответствующее обозначение.

Таблица 11.1. Обозначения уровней чистоты

№ вопроса	Обозначение	Описание
1	RNDS	Reads no database state – не читает базу данных
2	RNPS	Reads no package state – не читает пакет
3	WNDS	Writes no database state – не пишет в базу данных
4	WNPS	Writes no package state – не пишет в пакет

Следовательно, функция с уровнем чистоты {WNPS, WNDS} гарантированно не пишет в базу данных и не изменяет переменные пакета, но может ссылаться на переменные пакета и/или читать из таблиц базы данных. Чтобы функцию можно было вызывать из оператора SQL, ее уровень чистоты должен содержать хотя бы обозначение WNDS.

При использовании пакетов функций в версиях Oracle, предшествующих версии 8.1, требовалось указывать уровень чистоты, прежде чем вызывать функцию из оператора SQL. Для этого в спецификацию пакета добавлялся *псевдокомментарий* (*pragma*), или указание компилятору. Псевдокомментарий `RESTRICT_REFERENCES` в спецификации пакета следует за объявлением функции:

```
CREATE OR REPLACE PACKAGE my_pkg AS
  FUNCTION my_func(arg1 IN NUMBER) RETURN VARCHAR2;
  PRAGMA RESTRICT_REFERENCES(my_func, RNPS, WNPS, WNDS);
END my_pkg;
```



При компиляции тела пакета код проверяется на обозначения, перечисленные в псевдокомментарии `RESTRICT_REFERENCES`. Если код не соответствует объявленному в псевдокомментарии уровню чистоты, то компиляция не выполняется и выдается сообщение об ошибке:

```
PLS-00452: Subprogram 'MY_FUNC' violates its associated pragma
```

То есть в псевдокомментарии `RESTRICT_REFERENCES` вы говорите компилятору, что ваша функция будет делать, а чего – не будет, компилятор проверяет, говорите ли вы правду, и затем вы можете вызывать функцию любым способом, поддерживаемым данным уровнем чистоты, без дальнейшего вмешательства со стороны Oracle. Если бы функция не была включена в пакет, процессор Oracle не имел бы возможности проверить уровень чистоты функции до ее вызова и пришлось бы проверять логику функции на побочные эффекты при каждом ее вызове.



Возможность заявить уровень чистоты – это еще одна причина использования пакетов для программирования на PL/SQL. Для автономных процедур и функций заявить уровень чистоты невозможно.

Начиная с Oracle8i уже больше не обязательно указывать уровень чистоты функций в спецификации пакета. Если вы этого не сделаете, функции при каждом вызове из оператора SQL будут проверяться на соответствие минимальным требованиям. Однако когда есть такая возможность, следует включать псевдокомментарий в спецификацию пакета, чтобы код мог быть проверен при компиляции, а не проверялся бы при каждом выполнении.

## Верьте мне...

Одной из причин, по которой Oracle смягчила требования на объявление уровня чистоты в момент компиляции, является то, что PL/SQL может вызывать функции, написанные на C и Java, которые не имеют механизмов, подобных псевдокомментариям PL/SQL для объявления уровня чистоты. Чтобы позволить функциям, написанным на разных языках, вызывать друг друга, в Oracle8i добавлено ключевое слово `TRUST`. Добавление `TRUST` в псевдокомментарий `RESTRICT_REFERENCES` для функции приводит к тому, что Oracle:

1. Обрабатывает функцию как удовлетворяющую псевдокомментарий, не проводя проверки кода.
2. Обрабатывает все процедуры или функции, вызываемые из функции и имеющие ключевое слово `TRUST`, так, как если бы они тоже удовлетворяли псевдокомментарий.

Таким образом, хранимая функция, псевдокомментарий `RESTRICT_REFERENCES` которой содержит `WNDS` и `TRUST`, может вызывать другие функции PL/SQL, для которых не заданы псевдокомментарии `RESTRICT_RE-`

REFERENCES и/или внешние функции C и Java, и сама также может вызываться из операторов SQL. В случае внешнего вызова C или Java, если вы собираетесь вызывать функцию из SQL, необходимо для функции включить обозначение TRUST в псевдокомментарий RESTRICT\_REFERENCES, так как исходные тексты C и Java не доступны серверу для контроля.

Чтобы использовать ключевое слово TRUST, просто добавьте его в конец списка обозначений уровня чистоты:

```
CREATE OR REPLACE PACKAGE my_pkg AS
  FUNCTION my_func(arg1 IN NUMBER) RETURN VARCHAR2;
  PRAGMA RESTRICT_REFERENCES(my_func, RNPS, WNPS, WNDS, TRUST);
END my_pkg;
```

Хотя может показаться, что было бы удобно при объявлении уровня чистоты всегда применять TRUST, такой возможностью не стоит пользоваться часто. Если вы добавите в псевдокомментарий обозначение TRUST, дальнейшие изменения вашей функции или функций нижнего уровня, нарушающие WNDS, не будут обнаружены ни при компиляции, ни при выполнении, что приведет к непредвиденным результатам выполнения запросов.

## Другие ограничения

Помимо проверки условия WNDS, Oracle также проверяет, следует ли каждая функция, вызываемая из оператора SQL, перечисленным правилам:

1. Функция не может заканчивать текущую транзакцию, используя COMMIT или ROLLBACK.
2. Функция не может вносить изменения в транзакцию, создавая точки сохранения или откатываясь к ранее определенной точке сохранения.
3. Функция не может использовать оператор ALTER SYSTEM или ALTER SESSION.
4. Все типы параметров, включая возвращаемый тип, должны быть стандартными типами SQL, такими как VARCHAR2, NUMBER и DATE. Типы PL/SQL, такие как BOOLEAN и RECORD, типы коллекций, такие как VARRAY, и объектные типы не разрешены.

Три первых ограничения направлены на защиту от изменений, которые могли бы затронуть среду выполнения родительского запроса. Четвертое требование может быть смягчено в следующей версии сервера Oracle, но согласитесь, что сложно даже попытаться представить, как вызов функции, которая возвращает вложенную таблицу объектов, мог бы улучшить оператор SELECT.<sup>1</sup>

---

<sup>1</sup> Если только она не заключена в выражение TABLE инструкции FROM.



## Согласованность

Для вызова хранимой функции из запроса необходимо, чтобы были выполнены все рассмотренные ранее ограничения. Кроме того, существует еще одно дополнительное замечание. Это не столько ограничение, сколько подвох: запросы, выполняемые хранимыми функциями, почувствуют эффект транзакций, зафиксированных с момента начала выполнения родительского запроса, в то время как родительский запрос их не ощутит. Может быть, дело в модели данных, но так или иначе, что-то надо делать.

Например, если в 02-00 вы запускаете генерацию отчета по хранилищу данных, которое загружается между 14-00 и 16-00, то хранимые функции будут видеть те же данные, что и родительский запрос, если запрос будет завершать свое выполнение до загрузки следующих данных. С другой стороны, долго выполняющийся запрос над базой данных OLTP (Online Transaction Processing) в период пиковой активности может привести к серьезным противоречиям результатов, возвращенных родительским запросом и хранимыми функциями. Поэтому прежде чем включать в операторы SQL хранимые функции, внимательно проверьте свою операционную среду и предполагаемое время выполнения запроса.

## Хранимые функции в операторах DML

Хранимые функции также могут вызываться из операторов INSERT, UPDATE и DELETE. Большинство из описанных ранее ограничений относится и к хранимым функциям, вызываемым из операторов DML, но есть одно существенное отличие. Родительский оператор DML изменяет состояние базы данных, поэтому хранимые функции, вызываемые из операторов DML, не должны следовать ограничению WNDOS. Однако такие хранимые функции не могут читать или изменять ту же таблицу, что и родительский оператор DML.

Как и запросы, операторы DML могут вызывать хранимые функции отовсюду, где разрешены выражения, в том числе из:

- Инструкции VALUES оператора INSERT
- Инструкции SET оператора UPDATE
- Инструкции WHERE оператора INSERT, UPDATE или DELETE

Все подзапросы, вызываемые из оператора DML, также могут вызывать хранимые функции с теми же ограничениями, что и родительский оператор DML.

Часто наборы хранимых функций вызываются и из запросов, и из операторов DML. Например, ранее вы видели, как функция `pkg_util.translate_date` может вызываться из запроса для преобразования формата даты Oracle, хранящегося в базе данных, в формат, необходимый Ja-



ва-клиенту. Аналогично можно использовать перегруженную функцию `pkg_util.translate_date` в операторе `UPDATE` для выполнения обратного преобразования:

```
UPDATE cust_order
SET expected_ship_dt = pkg_util.translate_date(:1)
WHERE order_nbr = :2;
```

Здесь `:1` и `:2` – это заполнители для времени UTC и номера заказа, передаваемых Java-клиентом.

Хранимые функции могут использоваться в инструкции `WHERE` вместо связанных подзапросов, чтобы упростить операторы DML и облегчить повторное использование кода. Пусть, например, необходимо передвинуть предполагаемую дату отправки на пять дней для всех заказов, содержащих деталь с номером `F34-17802`. Можно написать оператор `UPDATE` для таблицы `cust_order`, используя связанный подзапрос:

```
UPDATE cust_order co
SET co.expected_ship_dt = NVL(co.expected_ship_dt, SYSDATE) + 5
WHERE co.cancelled_dt IS NULL and co.ship_dt IS NULL
AND EXISTS (SELECT 1 FROM line_item li
WHERE li.order_nbr = co.order_nbr
AND li.part_nbr = 'F34-17802');
```

Однако написав множество подзапросов к таблице `line_item`, вы поймете, что пришло время написать универсальную функцию и включить ее в пакет `pkg_util`:

```
FUNCTION get_part_count(ordno IN NUMBER,
partno IN VARCHAR2 DEFAULT NULL, max_cnt IN NUMBER DEFAULT 9999)
RETURN NUMBER IS
tot_cnt NUMBER(5) := 0;
li_part_nbr VARCHAR2(20);
CURSOR cur_li(c_ordno IN NUMBER) IS
SELECT part_nbr
FROM line_item
WHERE order_nbr = c_ordno;
BEGIN
OPEN cur_li(ordno);
WHILE tot_cnt < max_cnt LOOP
FETCH cur_li INTO li_part_nbr;
EXIT WHEN cur_li%NOTFOUND;

IF partno IS NULL OR
(partno IS NOT NULL AND partno = li_part_nbr) THEN
tot_cnt := tot_cnt + 1;
END IF;
END LOOP;
CLOSE cur_li;

RETURN tot_cnt;
END get_part_count;
```

Функцию можно использовать в разных целях, например:

1. Для подсчета количества пунктов определенного заказа.
2. Для подсчета количества пунктов определенного заказа, содержащих определенную деталь.
3. Чтобы определить, имеет ли определенный заказ как минимум X вхождений указанной детали.

Теперь оператор UPDATE может использовать функцию для поиска открытых заказов, которые содержат хотя бы одну деталь с номером F34-17802:

```
UPDATE cust_order co
SET co.expected_ship_dt = NVL(co.expected_ship_dt, SYSDATE) + 5
WHERE co.cancelled_dt IS NULL and co.ship_dt IS NULL
AND 1 = pkg_util.get_part_count(co.order_nbr, 'F34-17802', 1);
```

## SQL внутри PL/SQL

Вы узнали о том, как вызывать PL/SQL из SQL, теперь поговорим об обратном: как использовать SQL внутри PL/SQL-кода. SQL – это мощное средство манипулирования большими объемами данных, но бывают ситуации, когда необходимо работать с данными на уровне строки. PL/SQL с его возможностями управления курсором и организацией циклов обладает гибкостью при работе на уровне множеств с использованием SQL или на уровне строк с применением курсоров. Но многие PL/SQL-программисты отказываются от возможностей SQL и делают все на уровне строк, даже когда это не нужно и трудоемко.

Рассмотрим такую аналогию: представьте, что вы работаете на складе, и на погрузочную платформу прибывает большая партия деталей. Ваша задача – разделить партию по типам деталей и распределить их по разным участкам склада. Чтобы упростить работу, владелец склада приобрел лучший погрузчик, который можно было купить. Можно пойти двумя путями:

1. Брать по одной коробке, определять тип деталей и отвозить ее в соответствующее место назначения.
2. Потратить некоторое время на анализ ситуации, определить, что все коробки контейнера относятся к одному типу, и развезти по местам назначения целые контейнеры.

Хотя аналогия выглядит слишком упрощенно, она иллюстрирует разницу между операциями над множествами и над строками. Позволив серверу Oracle манипулировать большими наборами данных в одной операции, можно добиться увеличения производительности на несколько порядков по сравнению с манипулированием отдельными строками (особенно это касается систем с несколькими процессорами).



При использовании для доступа к базе данных процедурного языка (PL/SQL, C с вызовами OCI или Java, использующий JDBC) существует тенденция применять первую стратегию. Вероятно, программисты привыкли при использовании процедурного языка писать код на низком уровне детализации, и это распространяется и на их логику доступа к данным. Такая методика общепринята в системах, которые должны обрабатывать и загружать большие объемы данных из внешних файлов, например в утилитах загрузки хранилища данных.

Пусть необходимо построить инфраструктуру приема файлов от нескольких OLTP-систем (Online Transaction Processing), выполнения различных операций по очистке данных и обобщения данных в хранилище. Используя PL/SQL (или C, Java, C++, Cobol и т. д.), можно создать модули, выполняющие следующие действия:

1. Открытие заданного файла.
2. Чтение строки, проверку/очистку данных и обновление соответствующей строки реальной таблицы хранилища.
3. Повторение п. 2 до тех пор, пока файл не будет прочитан полностью.
4. Закрытие файла.

Для небольших файлов этот способ работает хорошо, но большие хранилища часто получают сотни тысяч или миллионы элементов данных. И даже если код невероятно эффективен, обработка файла, содержащего миллион строк, может занять несколько часов.

Существует альтернативный подход, использующий мощь сервера Oracle для обеспечения быстрой загрузки больших объемов данных:

1. Создать промежуточную таблицу для каждого уникального формата загружаемого файла.
2. В начале процесса загрузки выполнить отсечение для промежуточных таблиц.
3. Использовать SQL\*Loader, указав в параметрах прямой путь для быстрой загрузки файла данных в соответствующую промежуточную таблицу.
4. Обновить все строки промежуточной таблицы для очистки, проверки и преобразования данных, помечая строки как непригодные, если они не прошли проверку. Если возможно, параллельно выполнять операции для нескольких таблиц.
5. Обновить соответствующую реальную таблицу, используя подзапрос к промежуточной таблице. Опять-таки, если возможно, выполнять операции параллельно.

Для успешности этого подхода необходимо наличие соответствующего дискового пространства и достаточно большого табличного пространства для сегментов отката и временного хранения. Располагая соответствующими ресурсами и правильно построенными операторами SQL и



используя этот подход, можно получить десятикратное ускорение по сравнению с предыдущим способом.

Какую же роль в данном сценарии играет PL/SQL? PL/SQL замечательно подходит для выполнения шагов 4 и 5 предыдущего списка. Хотя хранимые процедуры могут содержать только один оператор обновления, SQL может быть сложным и содержать указания для оптимизатора и другие полезные функции. Поэтому рекомендуется изолировать SQL от остальной части приложения, чтобы его можно было независимо контролировать и настраивать.

В общем, при работе со сложной логикой, касающейся больших объемов данных, удобно думать в терминах множеств данных, а не шагов программирования. Другими словами, спросите себя о том, где находятся данные, куда они должны переместиться и что с ними должно произойти, вместо того чтобы думать о том, что должно произойти с каждой записью для удовлетворения бизнес-требований. Если вы пойдете по этому пути, то будете писать эффективные операторы SQL, использующие, где это нужно, PL/SQL, вместо того чтобы писать сложные подпрограммы PL/SQL, при необходимости применяющие SQL. Тем самым вы предоставите серверу возможность разбить большие объемы работ на множество элементов, выполняемых параллельно, что может значительно повысить производительность.

# 12

## Сложные групповые операции

Групповые операции обобщают данные нескольких строк. Инструкция `GROUP BY` и базовые групповые операции были представлены в главе 4. Системы принятия решений требуют применения более сложных групповых операций. Приложения хранилищ данных требуют обобщения различных аспектов данных. Для эффективной поддержки принятия решений необходимо обобщать данные транзакций на различных уровнях. В этой главе мы поговорим о сложных групповых операциях, используемых в системах принятия решений.

Oracle8i вводит несколько полезных расширений функциональных возможностей SQL по обобщению данных, в том числе:

- Функцию `ROLLUP` для вставки в результаты суммирования итогов и подытогов.
- Функцию `CUBE` для формирования подытогов для всех возможных комбинаций группировки столбцов.
- Функцию `GROUPING`, которая помогает корректно интерпретировать результаты, генерируемые функциями `ROLLUP` и `CUBE`.

В Oracle9i добавлена еще одна функция, генерирующая суммарную информацию заданного уровня, – `GROUPING SETS`.

### ROLLUP

В главе 4 было показано, как использовать инструкцию `GROUP BY` вместе с групповыми функциями для получения итоговых результатов. Например, если вы хотите вывести месячный объем продаж для каждого региона, можно выполнить следующий запрос:<sup>1</sup>

---

<sup>1</sup> Порядок столбцов в результирующих множествах, приведенных в этой главе, зависит от применяемой версии Oracle.

```

SELECT R.NAME REGION,
       TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH, SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID AND YEAR = 2001
GROUP BY R.NAME, O.MONTH;

```

REGION	MONTH	SUM(O.TOT_SALES)
Mid-Atlantic	January	610697
Mid-Atlantic	February	428678
Mid-Atlantic	March	637031
Mid-Atlantic	April	541146
Mid-Atlantic	May	592935
Mid-Atlantic	June	501485
Mid-Atlantic	July	606914
Mid-Atlantic	August	460520
Mid-Atlantic	September	392898
Mid-Atlantic	October	510117
Mid-Atlantic	November	532889
Mid-Atlantic	December	492458
New England	January	509215
New England	February	615746
New England	March	566483
New England	April	597622
New England	May	566285
New England	June	503354
New England	July	559334
New England	August	547656
New England	September	575589
New England	October	549648
New England	November	481395
New England	December	533314
SouthEast US	January	379021
SouthEast US	February	618423
SouthEast US	March	655993
SouthEast US	April	610017
SouthEast US	May	661094
SouthEast US	June	568572
SouthEast US	July	556992
SouthEast US	August	478765
SouthEast US	September	635211
SouthEast US	October	536841
SouthEast US	November	553866
SouthEast US	December	613700

36 rows selected.

Как и ожидалось, отчет выводит общий объем продаж для каждой комбинации регион-месяц. Однако в более сложных приложениях может потребоваться выводить подытог для каждого региона для всех месяцев и итог для всех регионов или подытог для всех регионов по каждому месяцу и итог для всех месяцев. То есть может возникнуть необходимость получения подытогов и итогов нескольких уровней.



## Использование UNION (старый способ)

В приложениях хранилищ данных часто требуется формирование суммарной информации по различным измерениям, а также вывод соответствующих подытогов и итогов. Генерирование и извлечение такого типа суммарных данных является главной целью практически всех информационных хранилищ.

Вы уже поняли, что простой запрос GROUP BY недостаточен для порождения подытогов и итогов, описанных в данном разделе. Чтобы осознать сложность задачи, попробуем написать запрос, который бы возвращал в одном выводе следующую информацию:

- Объем продаж за каждый месяц для каждого региона
- Подытоги для всех месяцев по каждому региону
- Общий объем продаж всех регионов за все месяцы

Можно сформировать несколько уровней итогов, используя запрос UNION (это единственный способ, доступный до появления Oracle8i). Например, следующий UNION-запрос выдаст три желаемых уровня подытогов:

```
SELECT R.NAME REGION,
       TO_CHAR(TO_DATE(0.MONTH, 'MM'), 'Month') MONTH, SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID AND YEAR = 2001
GROUP BY R.NAME, O.MONTH
UNION ALL
SELECT R.NAME REGION, NULL, SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID AND YEAR = 2001
GROUP BY R.NAME
UNION ALL
SELECT NULL, NULL, SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID AND YEAR = 2001;
```

REGION	MONTH	SUM(O.TOT_SALES)
Mid-Atlantic	January	610697
Mid-Atlantic	February	428676
Mid-Atlantic	March	637031
Mid-Atlantic	April	541146
Mid-Atlantic	May	592935
Mid-Atlantic	June	501485
Mid-Atlantic	July	606914
Mid-Atlantic	August	460520
Mid-Atlantic	September	392898
Mid-Atlantic	October	510117
Mid-Atlantic	November	532889
Mid-Atlantic	December	492458
New England	January	509215
New England	February	615746

New England	March	566483
New England	April	597622
New England	May	566285
New England	June	503354
New England	July	559334
New England	August	547656
New England	September	575589
New England	October	549648
New England	November	461395
New England	December	533314
SouthEast US	January	379021
SouthEast US	February	618423
SouthEast US	March	655993
SouthEast US	April	610017
SouthEast US	May	661094
SouthEast US	June	568572
SouthEast US	July	566992
SouthEast US	August	478765
SouthEast US	September	635211
SouthEast US	October	536841
SouthEast US	November	553866
SouthEast US	December	613700
Mid-Atlantic		6307766
New England		6585641
SouthEast US		6868495
		19761902

40 rows selected.

Запрос порождает 40 строк вывода, 36 из которых представляют объемы продаж за каждый месяц для каждого региона. Оставшиеся 4 строки – подытоги и итог. Три строки с названиями регионов и значениями NULL для месяца являются подытогами для каждого региона за все месяцы, а последняя строка со значениями NULL обоих столбцов – это общий объем продаж всех регионов за все месяцы.

Желаемый результат получен, теперь давайте проанализируем сам запрос. В данном примере таблица заказов очень мала, она содержит всего 720 строк. Итоговая информация требовалась только для двух измерений: региона и месяца. У нас было 3 региона и 12 месяцев. Чтобы получить из таблицы необходимый результат, мы написали запрос, состоящий из трех операторов SELECT, объединенных при помощи UNION ALL. Посмотрим план выполнения запроса:

Query Plan

```

-----
SELECT STATEMENT   Cost = 15
  UNION-ALL
    SORT GROUP BY
      HASH JOIN
        TABLE ACCESS FULL REGION
        TABLE ACCESS FULL ORDERS

```



```

SORT GROUP BY
  HASH JOIN
    TABLE ACCESS FULL REGION
    TABLE ACCESS FULL ORDERS
  SORT AGGREGATE
    NESTED LOOPS
      TABLE ACCESS FULL ORDERS
      INDEX UNIQUE SCAN PK7

```

14 rows selected.

Как показывает вывод EXPLAIN PLAN, для получения результатов Oracle должен выполнить следующие операции:

- Три полных сканирования таблицы заказов
- Два полных сканирования таблицы регионов
- Одно сканирование уникального ключа таблицы регионов (PK7)
- Два хеш-объединения
- Одно объединение NESTED LOOP
- Две операции SORT GROUP BY
- Одну операцию SORT AGGREGATE
- Одно слияние UNION ALL

В любом реальном приложении таблица заказов будет состоять из сотен тысяч строк, и выполнение всех этих операций потребует значительного времени. Еще хуже, если суммарная информация должна подготавливаться по большому количеству разрезов, тогда запрос придется еще усложнить. В результате такие запросы отрицательно влияют на производительность.

## Использование ROLLUP (новый способ)

Oracle8i вводит ряд новых возможностей для формирования нескольких уровней итоговой информации в одном запросе. Одним из таких нововведений является набор расширений инструкции GROUP BY. В Oracle8i добавлены два расширения для инструкции GROUP BY – ROLLUP и CUBE. Oracle9i вводит еще одно расширение – GROUPING SETS. Данный раздел будет посвящен ROLLUP. Операции CUBE и GROUPING SETS будут рассмотрены далее в этой же главе.

ROLLUP – это расширение инструкции GROUP BY, поэтому может появиться только в запросе, содержащем инструкцию GROUP BY. Операция ROLLUP группирует выделенные строки на основе выражений инструкции GROUP BY и подготавливает итоговую строку для каждой группы. Синтаксис ROLLUP таков:

```

SELECT ...
FROM ...
GROUP BY ROLLUP (упорядоченный список группируемых столбцов)

```



Используя ROLLUP, можно получить суммарную информацию, определенную в начале раздела, гораздо более простым способом, чем в запросе UNION ALL:

```
SELECT R.NAME REGION,
       TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH, SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID AND YEAR = 2001
GROUP BY ROLLUP (R.NAME, O.MONTH);
```

REGION	MONTH	SUM(O.TOT_SALES)
Mid-Atlantic	January	610697
Mid-Atlantic	February	428676
Mid-Atlantic	March	837031
Mid-Atlantic	April	541146
Mid-Atlantic	May	592935
Mid-Atlantic	June	501485
Mid-Atlantic	July	606914
Mid-Atlantic	August	460520
Mid-Atlantic	September	392898
Mid-Atlantic	October	510117
Mid-Atlantic	November	532889
Mid-Atlantic	December	492458
Mid-Atlantic		6307766
New England	January	509215
New England	February	615746
New England	March	566483
New England	April	597622
New England	May	586285
New England	June	503354
New England	July	559334
New England	August	547656
New England	September	575589
New England	October	549648
New England	November	461395
New England	December	533314
New England		6585641
SouthEast US	January	379021
SouthEast US	February	618423
SouthEast US	March	655993
SouthEast US	April	610017
SouthEast US	May	661094
SouthEast US	June	568572
SouthEast US	July	556992
SouthEast US	August	478765
SouthEast US	September	635211
SouthEast US	October	536841
SouthEast US	November	553866
SouthEast US	December	613700
SouthEast US		6868495
		19761902

40 rows selected.

Как видите, операция ROLLUP выводит подытоги для указанных разрезов и общий итог. Аргументом ROLLUP является упорядоченный список столбцов группировки. Так как операция ROLLUP используется совместно с инструкцией GROUP BY, она сначала генерирует обобщенные значения на основе операции GROUP BY над упорядоченным списком столбцов. Затем генерируются подытоги более высокого уровня и наконец – общий итог. ROLLUP не только упрощает запрос, но и улучшает производительность. Посмотрим на план выполнения запроса:

#### Query Plan

```
-----
SELECT STATEMENT  Cost = 7
  SORT GROUP BY ROLLUP
    HASH JOIN
      TABLE ACCESS FULL REGION
      TABLE ACCESS FULL ORDERS
```

Вместо того чтобы производить многочисленные сканирования таблиц, объединения и другие операции, необходимые в UNION-версии запроса, ROLLUP-запрос для получения желаемого результата требует всего одного полного просмотра таблицы регионов, одного полного просмотра таблицы заказов и одного объединения.

При желании получить подытоги за каждый месяц, а не для каждого региона, все, что нужно сделать, – это поменять порядок столбцов в операции ROLLUP:

```
SELECT R.NAME REGION,
       TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH, SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID AND YEAR = 2001
GROUP BY ROLLUP (O.MONTH, R.NAME);
```

REGION	MONTH	SUM(O.TOT_SALES)
Mid-Atlantic	January	610697
New England	January	509215
SouthEast US	January	379021
	January	1498933
Mid-Atlantic	February	428676
New England	February	615746
SouthEast US	February	618423
	February	1662845
Mid-Atlantic	March	637031
New England	March	566483
SouthEast US	March	655993
	March	1859507
Mid-Atlantic	April	541146
New England	April	597622
SouthEast US	April	610017
	April	1748785

Mid-Atlantic	May	592935
New England	May	566285
SouthEast US	May	661094
	May	1820314
Mid-Atlantic	June	501485
New England	June	503354
SouthEast US	June	568572
	June	1573411
Mid-Atlantic	July	606914
New England	July	559334
SouthEast US	July	556992
	July	1723240
Mid-Atlantic	August	460520
New England	August	547656
SouthEast US	August	478765
	August	1486941
Mid-Atlantic	September	392898
New England	September	575589
SouthEast US	September	635211
	September	1603698
Mid-Atlantic	October	510117
New England	October	549648
SouthEast US	October	536841
	October	1596606
Mid-Atlantic	November	532889
New England	November	461395
SouthEast US	November	553866
	November	1548150
Mid-Atlantic	December	492458
New England	December	533314
SouthEast US	December	613700
	December	1639472
		19761902

49 rows selected.

Введение дополнительных разрезов для суммирования не приводит к усложнению запроса. Следующий запрос выводит подытоги по региону, месяцу и году для первого квартала:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY ROLLUP (O.YEAR, O.MONTH, R.NAME);
```

YEAR MONTH	REGION	SUM(O.TOT_SALES)
-----		
2000 January	Mid-Atlantic	1221394
2000 January	New England	1018430
2000 January	SouthEast US	758042



2000	January		2997866
2000	February	Mid-Atlantic	857352
2000	February	New England	1231492
2000	February	SouthEast US	1236846
2000	February		3325690
2000	March	Mid-Atlantic	1274062
2000	March	New England	1132966
2000	March	SouthEast US	1311986
2000	March		3719014
2000			10042570
2001	January	Mid-Atlantic	610697
2001	January	New England	509215
2001	January	SouthEast US	379021
2001	January		1498933
2001	February	Mid-Atlantic	428876
2001	February	New England	615746
2001	February	SouthEast US	618423
2001	February		1662845
2001	March	Mid-Atlantic	637031
2001	March	New England	566483
2001	March	SouthEast US	655993
2001	March		1859507
2001			5021285
			15063855

27 rows selected.

## Частичная операция ROLLUP

В ROLLUP-запросе с N измерениями (разрезами) общий итог рассматривается как самый верхний уровень. Строки различных подытогов измерения N-1 составляют следующий более низкий уровень, строки подытогов измерения N-2 – еще более низкий уровень, и т. д. В последнем примере у нас было три размерности (год, месяц и регион), и строка общих итогов представляла собой верхний уровень. Подытоги за год – это более низкий уровень, так как эти строки являются подытогами для двух измерений (месяц и регион). Строки подытогов для комбинации год-месяц находятся еще на уровень ниже, являясь подытогами для одного измерения (региона). Остальные строки являются результатом обычной операции GROUP BY (без ROLLUP) и образуют самый низкий уровень.

При желании исключить некоторые подытоги и итоги из вывода ROLLUP можно двигаться только сверху вниз, то есть сначала исключить общий итог, потом постепенно двигаться к подытогам более низких уровней. Для выполнения этой операции необходимо убрать один или более столбцов из операции ROLLUP и поместить их в инструкцию GROUP BY. Получится так называемый *частичный ROLLUP* (*partial ROLLUP*).

В качестве примера частичной операции ROLLUP давайте посмотрим, что произойдет, если убрать из предыдущей операции ROLLUP первый столбец O.YEAR и перенести его в инструкцию GROUP BY.

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY O.YEAR, ROLLUP (O.MONTH, R.NAME);
```

YEAR	MONTH	REGION	SUM(O.TOT_SALES)
-----			
2000	January	Mid-Atlantic	1221394
2000	January	New England	1018430
2000	January	SouthEast US	758042
2000	January		2997866
2000	February	Mid-Atlantic	857352
2000	February	New England	1231492
2000	February	SouthEast US	1236846
2000	February		3325690
2000	March	Mid-Atlantic	1274062
2000	March	New England	1132966
2000	March	SouthEast US	1311986
2000	March		3719014
2000			10042570
2001	January	Mid-Atlantic	610687
2001	January	New England	509215
2001	January	SouthEast US	379021
2001	January		1498933
2001	February	Mid-Atlantic	428676
2001	February	New England	615746
2001	February	SouthEast US	618423
2001	February		1662845
2001	March	Mid-Atlantic	637031
2001	March	New England	566483
2001	March	SouthEast US	655993
2001	March		1859507
2001			5021285

26 rows selected.

Данный запрос исключает из вывода общий итог. Убрав столбец O.YEAR из операции ROLLUP, вы просите базу данных не накапливать суммарную информацию по годам. Следовательно, информация будет обобщаться по месяцу и региону. Если затем удалить из операции ROLLUP столбец O.MONTH, запрос не будет накапливать информацию по месяцу и в выводе будут присутствовать только подытоги по региону:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
```

```
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY O.YEAR, O.MONTH, ROLLUP (R.NAME);
```

YEAR	MONTH	REGION	SUM(O.TOT_SALES)
2000	January	Mid-Atlantic	1221394
2000	January	New England	1018430
2000	January	SouthEast US	758042
2000	January		2997866
2000	February	Mid-Atlantic	857352
2000	February	New England	1231492
2000	February	SouthEast US	1236846
2000	February		3325690
2000	March	Mid-Atlantic	1274062
2000	March	New England	1132966
2000	March	SouthEast US	1311986
2000	March		3719014
2001	January	Mid-Atlantic	810697
2001	January	New England	509215
2001	January	SouthEast US	379021
2001	January		1498933
2001	February	Mid-Atlantic	428676
2001	February	New England	815746
2001	February	SouthEast US	818423
2001	February		1662845
2001	March	Mid-Atlantic	637031
2001	March	New England	566483
2001	March	SouthEast US	855993
2001	March		1859507

24 rows.

## CUBE

С расширением CUBE инструкции GROUP BY обобщение делает шаг вперед относительно ROLLUP. Операция CUBE генерирует подытоги для всех возможных комбинаций столбцов группировки. Следовательно, вывод операции CUBE будет содержать все подытоги аналогичной операции ROLLUP, а также некоторые дополнительные. Например, если вы выполняете ROLLUP для столбцов региона и месяца, то получите подытоги для всех месяцев по каждому региону и общий итог. Если же выполнить соответствующую операцию CUBE, то будут выведены:

- Обычные строки, порожденные инструкцией GROUP BY
- Подытоги для всех месяцев по каждому региону
- Подытоги для всех регионов по каждому месяцу
- Общий итог



Как и ROLLUP, CUBE – это расширение инструкции GROUP BY и может использоваться только в запросах, содержащих эту инструкцию. Синтаксис CUBE таков:

```
SELECT ...
FROM ...
GROUP BY CUBE (список группируемых столбцов)
```

Например, следующий запрос возвращает подытоги для всех комбинаций регионов и месяцев таблицы ORDER:

```
SELECT R.NAME REGION, TO_CHAR(TO_DATE(0.MONTH, 'NN'), 'Month') MONTH,
SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
GROUP BY CUBE(R.NAME, O.MONTH);
```

REGION	MONTH	SUM(O.TOT_SALES)
Mid-Atlantic	January	1832091
Mid-Atlantic	February	1286028
Mid-Atlantic	March	1911093
Mid-Atlantic	April	1623438
Mid-Atlantic	May	1778805
Mid-Atlantic	June	1504455
Mid-Atlantic	July	1820742
Mid-Atlantic	August	1381560
Mid-Atlantic	September	1178694
Mid-Atlantic	October	1530351
Mid-Atlantic	November	1598667
Mid-Atlantic	December	1477374
Mid-Atlantic		18923298
New England	January	1527645
New England	February	1847238
New England	March	1699449
New England	April	1792866
New England	May	1698855
New England	June	1510062
New England	July	1678002
New England	August	1642968
New England	September	1726767
New England	October	1648944
New England	November	1384185
New England	December	1599942
New England		19756923
SouthEast US	January	1137063
SouthEast US	February	1855269
SouthEast US	March	1967979
SouthEast US	April	1830051
SouthEast US	May	1983282
SouthEast US	June	1705716
SouthEast US	July	1670976
SouthEast US	August	1436295

SouthEast US	September	1905633
SouthEast US	October	1610523
SouthEast US	November	1661598
SouthEast US	December	1841100
SouthEast US		20605485
	January	4496799
	February	4988535
	March	5578521
	April	5246355
	May	5460942
	June	4720233
	July	5169720
	August	4460823
	September	4811094
	October	4789818
	November	4644450
	December	4918416
		59285706

52 rows selected.

Обратите внимание, что вывод содержит не только подытоги для каждого региона, но и подытоги для каждого месяца. Можно получить такой же результат от запроса, не применяя операцию CUBE. Однако новый запрос будет длиннее и сложнее, чем первый, и, конечно же, он менее эффективен. Выглядеть новый запрос может, например, так:

```

SELECT R.NAME REGION, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
GROUP BY R.NAME, O.MONTH
UNION ALL
SELECT R.NAME REGION, NULL, SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
GROUP BY R.NAME
UNION ALL
SELECT NULL, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH, SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
GROUP BY O.MONTH
UNION ALL
SELECT NULL, NULL, SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID;

```

REGION	MONTH	SUM(O.TOT_SALES)
-----	-----	-----
Mid-Atlantic	January	1832091
Mid-Atlantic	February	1286028
Mid-Atlantic	March	1911093

Mid-Atlantic	April	1623438
Mid-Atlantic	May	1778805
Mid-Atlantic	June	1504455
Mid-Atlantic	July	1820742
Mid-Atlantic	August	1381560
Mid-Atlantic	September	1178694
Mid-Atlantic	October	1530351
Mid-Atlantic	November	1598667
Mid-Atlantic	December	1477374
New England	January	1527645
New England	February	1847238
New England	March	1699449
New England	April	1792866
New England	May	1898855
New England	June	1510062
New England	July	1678002
New England	August	1642968
New England	September	1726767
New England	October	1648944
New England	November	1384185
New England	December	1599942
SouthEast US	January	1137063
SouthEast US	February	1855269
SouthEast US	March	1967979
SouthEast US	April	1830051
SouthEast US	May	1983282
SouthEast US	June	1705716
SouthEast US	July	1670976
SouthEast US	August	1436295
SouthEast US	September	1905633
SouthEast US	October	1610523
SouthEast US	November	1661598
SouthEast US	December	1841100
Mid-Atlantic		18923298
New England		19756923
SouthEast US		20605485
	January	4496799
	February	4988535
	March	5578521
	April	5246355
	May	5460942
	June	4720233
	July	5169720
	August	4460823
	September	4811094
	October	4789818
	November	4644450
	December	4918416
		59285706

52 rows selected.



Так как CUBE выводит обобщенные результаты для всех возможных комбинаций группируемых столбцов, вывод запроса, использующего CUBE, не зависит от порядка столбцов в операции CUBE (если ничто, кроме порядка, не изменяется). Для ROLLUP дело обстоит не так. Если все остальное в запросе остается как прежде, ROLLUP(a,b) выведет результирующее множество, отличающееся от результата ROLLUP(b,a). А результирующее множество CUBE(a,b) совпадает с результатом CUBE(b,a). Для наглядности возьмем пример из начала раздела и поменяем порядок столбцов в операции CUBE:

```
SELECT R.NAME REGION, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
GROUP BY CUBE(O.MONTH, R.NAME);
```

REGION	MONTH	SUM(O.TOT_SALES)
Mid-Atlantic	January	1832091
New England	January	1527645
SouthEast US	January	1137063
	January	4496799
Mid-Atlantic	February	1286028
New England	February	1847238
SouthEast US	February	1855269
	February	4988535
Mid-Atlantic	March	1911093
New England	March	1699449
SouthEast US	March	1967979
	March	5578521
Mid-Atlantic	April	1623438
New England	April	1792868
SouthEast US	April	1830051
	April	5246355
Mid-Atlantic	May	1778805
New England	May	1698855
SouthEast US	May	1983282
	May	5460942
Mid-Atlantic	June	1504455
New England	June	1510062
SouthEast US	June	1705716
	June	4720233
Mid-Atlantic	July	1820742
New England	July	1678002
SouthEast US	July	1670976
	July	5169720
Mid-Atlantic	August	1381560
New England	August	1642968
SouthEast US	August	1436295
	August	4460823
Mid-Atlantic	September	1178694

New England	September	1726767
SouthEast US	September	1905633
	September	4811094
Mid-Atlantic	October	1530351
New England	October	1848944
SouthEast US	October	1810523
	October	4789818
Mid-Atlantic	November	1598667
New England	November	1384185
SouthEast US	November	1661598
	November	4644450
Mid-Atlantic	December	1477374
New England	December	1599942
SouthEast US	December	1841100
	December	4918416
Mid-Atlantic		18923298
New England		19756923
SouthEast US		20605485
		59285706

52 rows selected.

Запрос вывел те же результаты, что и предыдущий, изменился только порядок строк.

Чтобы исключить из вывода некоторые подытоги, можно выполнить *частичную (partial)* операцию CUBE (аналогично частичной операции ROLLUP), убрав столбец (или столбцы) из операции CUBE и поместив его (их) в инструкцию GROUP BY, например:

```
SELECT R.NAME REGION, TO_CHAR(TO_DATE(0.MONTH, 'MM'), 'Month') MONTH,
SUM(0.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
GROUP BY R.NAME CUBE(0.MONTH);
```

REGION	MONTH	SUM(0.TOT_SALES)
Mid-Atlantic	January	1832091
Mid-Atlantic	February	1286028
Mid-Atlantic	March	1911093
Mid-Atlantic	April	1623438
Mid-Atlantic	May	1778805
Mid-Atlantic	June	1504455
Mid-Atlantic	July	1820742
Mid-Atlantic	August	1381560
Mid-Atlantic	September	1178694
Mid-Atlantic	October	1530351
Mid-Atlantic	November	1598667
Mid-Atlantic	December	1477374
Mid-Atlantic		18923298
New England	January	1527645
New England	February	1847238

New England	March	1699449
New England	April	1792866
New England	May	1698855
New England	June	1510062
New England	July	1678002
New England	August	1642988
New England	September	1726767
New England	October	1648944
New England	November	1384185
New England	December	1599942
New England		19756923
SouthEast US	January	1137063
SouthEast US	February	1855269
SouthEast US	March	1987979
SouthEast US	April	1830051
SouthEast US	May	1983282
SouthEast US	June	1705716
SouthEast US	July	1670976
SouthEast US	August	1436295
SouthEast US	September	1905633
SouthEast US	October	1610523
SouthEast US	November	1661598
SouthEast US	December	1841100
SouthEast US		20605485

39 rows selected.

Если вы сравните результаты частичной операции CUBE с результатами полной операции CUBE, представленной в начале раздела, то заметите, что частичный CUBE исключает из вывода подытоги для каждого месяца и общий итог. Если нужно оставить подытоги для каждого месяца, но исключить подытоги для каждого региона, поменяйте местами R.NAME и O.MONTH в инструкции GROUP BY... CUBE:

```
SELECT R.NAME REGION, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
GROUP BY O.MONTH CUBE(R.NAME);
```

Интересно, что если операция CUBE содержит один столбец, то она выводит тот же результат, что и операция ROLLUP. Поэтому два приведенные ниже запроса выводят идентичные результаты:

```
SELECT R.NAME REGION, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
GROUP BY R.NAME CUBE(O.MONTH);

SELECT R.NAME REGION, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       SUM(O.TOT_SALES)
```



```
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
GROUP BY R.NAME ROLLUP(O.MONTH);
```

## Функция GROUPING

ROLLUP и CUBE выводят дополнительные строки, содержащие подытоги и итоги. Эти строки в одном или более столбцах содержат значения NULL. Вывод, содержащий значения NULL и представляющий подытоги, будет непонятен человеку, который не знаком с поведением операций ROLLUP и CUBE. Разве вашего вице-президента должно интересовать, чем именно вы пользуетесь (ROLLUP и CUBE или какими-то другими операциями), составляя ему ежемесячный отчет о продажах по регионам? Конечно же, нет. Именно поэтому эти страницы читаете вы, а не ваш начальник.

Если вы помните обходной маневр, обеспечиваемый функцией NVL, то, вероятно, попытаетесь преобразовать каждое значение NULL, полученное от ROLLUP и CUBE, в некоторое описательное значение, например:

```
SELECT NVL(TO_CHAR(O.YEAR), 'All Years') YEAR,
       NVL(TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month'), 'First Quarter') MONTH,
       NVL(R.NAME, 'All Regions') REGION, SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY ROLLUP (O.YEAR, O.MONTH, R.NAME);
```

YEAR	MONTH	REGION	SUM(O.TOT_SALES)
2000	January	Mid-Atlantic	1221394
2000	January	New England	1018430
2000	January	SouthEast US	758042
2000	January	All Regions	2997866
2000	February	Mid-Atlantic	857352
2000	February	New England	1231492
2000	February	SouthEast US	1236846
2000	February	All Regions	3325690
2000	March	Mid-Atlantic	1274062
2000	March	New England	1132966
2000	March	SouthEast US	1311986
2000	March	All Regions	3719014
2000	First Quarter	All Regions	10042570
2001	January	Mid-Atlantic	610697
2001	January	New England	509215
2001	January	SouthEast US	379021
2001	January	All Regions	1498933
2001	February	Mid-Atlantic	428676
2001	February	New England	615746
2001	February	SouthEast US	618423
2001	February	All Regions	1662845

2001	March	Mid-Atlantic	637031
2001	March	New England	566483
2001	March	SouthEast US	655993
2001	March	All Regions	1859507
2001	First Quarter	All Regions	5021285
All Years	First Quarter	All Regions	15063855

27 rows selected.

Функция NVL в данном примере сработала отлично. Однако если сами данные содержат значения NULL, то нет никакой возможности отличить значение NULL, представляющее отсутствующие данные, от столбца строки подытогов. И тогда использование функции NVL может вызывать проблемы. Рассмотрим пример, иллюстрирующий вышесказанное:

```
SELECT * FROM CUST_ORDER;
```

ORDER_NBR	CUST_NBR	SALES_EMP_ID	SALE_PRICE	ORDER_DT	EXPECTED	STATUS
1001	231	7354	99	22-JUL-01	23-JUL-01	DELIVERED
1000	201	7354	19	19-JUL-01	24-JUL-01	
1002	255	7368	12	12-JUL-01	25-JUL-01	
1003	264	7368	56	16-JUL-01	26-JUL-01	DELIVERED
1004	244	7368	34	18-JUL-01	27-JUL-01	PENDING
1005	288	7368	99	22-JUL-01	24-JUL-01	DELIVERED
1006	231	7354	22	22-JUL-01	28-JUL-01	
1007	255	7368	25	20-JUL-01	22-JUL-01	PENDING
1008	255	7368	25	21-JUL-01	23-JUL-01	PENDING
1009	231	7354	56	18-JUL-01	22-JUL-01	DELIVERED
1012	231	7354	99	22-JUL-01	23-JUL-01	DELIVERED
1011	201	7354	19	19-JUL-01	24-JUL-01	
1015	255	7368	12	12-JUL-01	25-JUL-01	
1017	264	7368	56	16-JUL-01	26-JUL-01	DELIVERED
1019	244	7368	34	18-JUL-01	27-JUL-01	PENDING
1021	288	7368	99	22-JUL-01	24-JUL-01	DELIVERED
1023	231	7354	22	22-JUL-01	28-JUL-01	
1025	255	7368	25	20-JUL-01	22-JUL-01	PENDING
1027	255	7368	25	21-JUL-01	23-JUL-01	PENDING
1029	231	7354	56	18-JUL-01	22-JUL-01	DELIVERED

20 rows selected.

Как видите, столбец STATUS содержит значения NULL. Если вы хотите просуммировать статусы заказов для каждого клиента и выполняете следующий запрос (обратите внимание на применение функции NVL к столбцу STATUS), то полученный результат может вас удивить.

```
SELECT NVL(TO_CHAR(CUST_NBR), 'All Customers') CUSTOMER,
       NVL(STATUS, 'All Status') STATUS,
       COUNT(*) FROM CUST_ORDER
GROUP BY CUBE(CUST_NBR, STATUS);
```

CUSTOMER	STATUS	COUNT(*)
201	All Status	2
201	All Status	2
231	DELIVERED	4
231	All Status	2
231	All Status	6
244	PENDING	2
244	All Status	2
255	PENDING	4
255	All Status	2
255	All Status	6
264	DELIVERED	2
264	All Status	2
288	DELIVERED	2
288	All Status	2
All Customers	DELIVERED	8
All Customers	PENDING	8
All Customers	All Status	6
All Customers	All Status	20

18 rows selected.

В полученном результате нет никакого смысла. Отправив такой отчет начальнику, вы сделаете себя кандидатом на увольнение. Проблема в том, что каждый раз, когда столбец STATUS содержит значение NULL, функция NVL возвращает строку «All Status». Очевидно, что в данной ситуации использование NVL не приносит пользы. Но не беспокойтесь! Oracle8i предоставляет решение данной проблемы – функцию GROUPING.

Функция GROUPING используется только в сочетании с операцией ROLLUP или CUBE. Функция GROUPING принимает в качестве входного параметра название столбца группировки и возвращает 0 или 1. Значение 1 возвращается, если в результате обобщения (ROLLUP или CUBE) появляется NULL; иначе возвращается 0. Стандартный синтаксис функции GROUPING таков:

```
SELECT ... [GROUPING(имя_столбца_группировки)] ...
FROM ...
GROUP BY ... {ROLLUP | CUBE} (имя_столбца_группировки)
```

Следующий пример демонстрирует использование функции GROUPING, просто возвращая ее результаты для трех столбцов, переданных ROLLUP:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES),
       GROUPING(O.YEAR) Y, GROUPING(O.MONTH) M, GROUPING(R.NAME) R
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY ROLLUP (O.YEAR, O.MONTH, R.NAME);
```



YEAR	MONTH	REGION	SUM(O.TOT_SALES)	Y	M	R
2000	January	Mid-Atlantic	1221394	0	0	0
2000	January	New England	1018430	0	0	0
2000	January	SouthEast US	758042	0	0	0
2000	January		2997866	0	0	1
2000	February	Mid-Atlantic	857352	0	0	0
2000	February	New England	1231492	0	0	0
2000	February	SouthEast US	1236846	0	0	0
2000	February		3325690	0	0	1
2000	March	Mid-Atlantic	1274062	0	0	0
2000	March	New England	1132966	0	0	0
2000	March	SouthEast US	1311986	0	0	0
2000	March		3719014	0	0	1
2000			10042570	0	1	1
2001	January	Mid-Atlantic	610697	0	0	0
2001	January	New England	509215	0	0	0
2001	January	SouthEast US	379021	0	0	0
2001	January		1498933	0	0	1
2001	February	Mid-Atlantic	428676	0	0	0
2001	February	New England	615746	0	0	0
2001	February	SouthEast US	618423	0	0	0
2001	February		1662845	0	0	1
2001	March	Mid-Atlantic	637031	0	0	0
2001	March	New England	566483	0	0	0
2001	March	SouthEast US	655993	0	0	0
2001	March		1859507	0	0	1
2001			5021285	0	1	1
			15063855	1	1	1

27 rows selected.

Посмотрите на столбцы вывода Y, M и R. Строка 4 – это подытог уровня региона для определенного месяца и года, поэтому функция GROUPING выдает значение 1 для региона и 0 – для месяца и года. Строка 26 (предпоследняя) – это подытог по всем регионам и месяцам для определенного года, следовательно, функция GROUPING выдает значение 1 для месяца и региона и 0 – для года. Строка 27 (общий итог) содержит 1 для всех столбцов группировки.

Комбинируя GROUPING и DECODE, можно создать более удобочитаемый вывод запроса, использующего CUBE и ROLLUP, например:

```
SELECT DECODE(GROUPING(O.YEAR), 1, 'All Years', O.YEAR) Year,
       DECODE(GROUPING(O.MONTH), 1, 'All Months',
              TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month')) Month,
       DECODE(GROUPING(R.NAME), 1, 'All Regions', R.NAME) Region, SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY ROLLUP (O.YEAR, O.MONTH, R.NAME);
```

YEAR	MONTH	REGION	SUM(0. TOT_SALES)
2000	January	Mid-Atlantic	1221394
2000	January	New England	1018430
2000	January	SouthEast US	758042
2000	January	All Regions	2997866
2000	February	Mid-Atlantic	857352
2000	February	New England	1231492
2000	February	SouthEast US	1238846
2000	February	All Regions	3325890
2000	March	Mid-Atlantic	1274062
2000	March	New England	1132966
2000	March	SouthEast US	1311986
2000	March	All Regions	3719014
2000	All Months	All Regions	10042570
2001	January	Mid-Atlantic	610697
2001	January	New England	509215
2001	January	SouthEast US	379021
2001	January	All Regions	1498933
2001	February	Mid-Atlantic	428676
2001	February	New England	615746
2001	February	SouthEast US	618423
2001	February	All Regions	1662845
2001	March	Mid-Atlantic	637031
2001	March	New England	566483
2001	March	SouthEast US	655993
2001	March	All Regions	1859507
2001	All Months	All Regions	5021285
All Years	All Months	All Regions	15063855

27 rows selected.

Используя DECODE вместе с GROUPING, вы получаете такой же результат, который получился при применении NVL в начале раздела. Однако благодаря использованию GROUPING и DECODE, нет риска неправильной интерпретации значений NULL в строках суммирования. В следующем примере видно, что функция GROUPING по-разному трактует значения NULL в строках итога и подытога и значения NULL в строках суммирования:

```
SELECT DECODE(GROUPING(CUST_NBR), 1, 'All Customers', CUST_NBR) CUSTOMER,
       DECODE(GROUPING(STATUS), 1, 'All Status', STATUS) STATUS, COUNT(*)
FROM CUST_ORDER
GROUP BY CUBE(CUST_NBR, STATUS);
```

CUSTOMER	STATUS	COUNT(*)
201		2
201	All Status	2
231	DELIVERED	4
231		2
231	All Status	6
244	PENDING	2

244	All Status	2
255	PENDING	4
255		2
255	All Status	6
264	DELIVERED	2
264	All Status	2
288	DELIVERED	2
288	All Status	2
All Customers	DELIVERED	8
All Customers	PENDING	6
All Customers		6
All Customers	All Status	20

18 rows selected.



Oracle9i вводит две новые функции, связанные с GROUPING: GROUPING\_ID и GROUP\_ID, которые описаны далее в разделе «Возможности группировки в Oracle 9i». Тем, кто работает с Oracle9i, стоит познакомиться с этими функциями.

## GROUPING SETS

Вы уже видели, как можно обобщать информацию при помощи ROLLUP и CUBE. Однако вывод ROLLUP и CUBE включает строки итогов и строки, порождаемые обычной операцией GROUP BY. Oracle9i вводит еще одно расширение инструкции GROUP BY — GROUPING SETS, используя которое, можно генерировать суммарную информацию только необходимого уровня, не включая в вывод строки, полученные в результате выполнения обычной операции GROUP BY.

Аналогично ROLLUP и CUBE, GROUPING SETS является расширением инструкции GROUP BY и может присутствовать только в запросе, содержащем GROUP BY. Синтаксис GROUP BY таков:

```
SELECT ...
FROM ...
GROUP BY GROUPING SETS (список столбцов группировки)
```

Давайте рассмотрим пример, чтобы лучше понять, как работает GROUPING SETS:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY GROUPING SETS (O.YEAR, O.MONTH, R.NAME);
```

YEAR MONTH	REGION	SUM(O.TOT_SALES)
2000		10042570



2001		5021285
January		4496799
February		4988535
March		5578521
	Mid-Atlantic	5029212
	New England	5074332
	SouthEast US	4960311

8 rows selected.

Обратите внимание, что вывод содержит только подытоги уровня региона, месяца и года, а обычных, более подробных данных GROUP BY в нем нет. Порядок столбцов операции GROUPING SETS не имеет значения. Операция выводит один и тот же результат вне зависимости от порядка столбцов, меняется только порядок строк вывода (он повторяет последовательность столбцов операции GROUPING). Например, если изменить порядок столбцов (O.YEAR, O.MONTH, R.NAME) на (O.MONTH, R.NAME, O.YEAR), то первыми будут выведены строки итоговой информации для месяца, потом для региона и последними – для года:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY GROUPING SETS (O.MONTH, R.NAME, O.YEAR);
```

YEAR MONTH	REGION	SUM(O.TOT_SALES)
January		4496799
February		4988535
March		5578521
	Mid-Atlantic	5029212
	New England	5074332
	SouthEast US	4960311
2000		10042570
2001		5021285

8 rows selected.

## Возможности группировки в Oracle9i

Рассмотренные ранее примеры представляют простые способы обобщения данных при помощи расширений инструкции GROUP BY. Oracle9i предоставляет возможность суммирования данных по более сложным правилам. В следующих разделах будут подробно описаны:

- Повторение названий столбцов в инструкции GROUP BY
- Группировка по составным столбцам
- Каскадная группировка
- Функции GROUPING\_ID и GROUP\_ID

## Повторение названий столбцов в инструкции GROUP BY

В Oracle8i повторение названий столбцов в инструкции GROUP BY не разрешено. Если инструкция GROUP BY содержит расширение (ROLLUP или CUBE), нельзя использовать один и тот же столбец внутри и вне расширения. Следующий оператор SQL в Oracle8i недопустим и приведет к выводу сообщения об ошибке:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY O.YEAR, ROLLUP (O.YEAR, O.MONTH, R.NAME);
GROUP BY O.YEAR, ROLLUP (O.YEAR, O.MONTH, R.NAME)
```

ERROR at line 6:

ORA-30490: Ambiguous expression in GROUP BY ROLLUP or CUBE list

Но в Oracle9i тот же самый оператор работает:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) TOTAL
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY O.YEAR, ROLLUP (O.YEAR, O.MONTH, R.NAME);
```

YEAR	MONTH	REGION	TOTAL
2000	January	Mid-Atlantic	1221394
2000	January	New England	1018430
2000	January	SouthEast US	758042
2000	January		2997866
2000	February	Mid-Atlantic	857352
2000	February	New England	1231492
2000	February	SouthEast US	1236546
2000	February		3325890
2000	March	Mid-Atlantic	1274062
2000	March	New England	1132966
2000	March	SouthEast US	1311986
2000	March		3719014
2001	January	Mid-Atlantic	610697
2001	January	New England	509215
2001	January	SouthEast US	379021
2001	January		1498933
2001	February	Mid-Atlantic	428676
2001	February	New England	615746
2001	February	SouthEast US	618423
2001	February		1662845

2001 March	Mid-Atlantic	637031
2001 March	New England	566483
2001 March	SouthEast US	655993
2001 March		1859507
2000		10042570
2001		5021285
2000		10042570
2001		5021285

28 rows selected.

Использование столбца O.YEAR в инструкции GROUP BY и в операции ROLLUP приводит к повторному выводу суммарных строк для каждого года и скрытию общего итога. Повторение названий столбцов в инструкции GROUP BY не слишком полезно, но все же стоит знать, что в Oracle9i подобные конструкции разрешены.

## Группировка по составным столбцам

Oracle8i поддерживает только группировки по отдельным столбцам. Oracle9i распространяет операции группировки и на составные столбцы. *Составной столбец (composite column)* – это набор из двух или более столбцов, значения которых при выполнении групповых расчетов рассматриваются как единое целое. Oracle8i разрешает групповые операции типа ROLLUP (a, b, c), в то время как Oracle9i также допускает применение групповых операций, подобных ROLLUP (a, (b, c)). В этом случае групповые расчеты воспринимают (b, c) как один столбец, например:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY ROLLUP ((O.YEAR, O.MONTH), R.NAME);
```

YEAR	MONTH	REGION	TOTAL
2000	January	Mid-Atlantic	1221394
2000	January	New England	1018430
2000	January	SouthEast US	758042
2000	January		2997866
2000	February	Mid-Atlantic	857352
2000	February	New England	1231492
2000	February	SouthEast US	1236846
2000	February		3325690
2000	March	Mid-Atlantic	1274062
2000	March	New England	1132966
2000	March	SouthEast US	1311986
2000	March		3719014
2001	January	Mid-Atlantic	610697
2001	January	New England	509215



2001 January	SouthEast US	379021
2001 January		1498933
2001 February	Mid-Atlantic	428676
2001 February	New England	615746
2001 February	SouthEast US	618423
2001 February		1662845
2001 March	Mid-Atlantic	637031
2001 March	New England	566483
2001 March	SouthEast US	655993
2001 March		1859507
		15083855

25 rows selected.

В данном примере два столбца (O.YEAR, O.MONTH) рассматриваются как один составной столбец. Oracle трактует комбинацию месяц-год как одно измерение и соответствующим образом генерирует суммарные строки. Хотя такой запрос недопустим в Oracle8i, можно имитировать группировку по составному столбцу, используя оператор конкатенации || для объединения двух столбцов и рассматривая результат как один составной столбец. Тогда Oracle8i может вывести такой же результат, что и предыдущий запрос на Oracle 9i, например:

```
SELECT TO_CHAR(O.YEAR)||' '||TO_CHAR(TO_DATE(O.MONTH,'MM'),'Month')
       Year_Month,
       R.NAME REGION, SUM(O.TOT_SALES)
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY
ROLLUP (TO_CHAR(O.YEAR)||' '||TO_CHAR(TO_DATE(O.MONTH,'MM'),'Month'), R. NAME);
```

YEAR_MONTH	REGION	SUM(O.TOT_SALES)
2000 February	Mid-Atlantic	857352
2000 February	New England	1231492
2000 February	SouthEast US	1236846
2000 February		3325690
2000 January	Mid-Atlantic	1221394
2000 January	New England	1018430
2000 January	SouthEast US	758042
2000 January		2997866
2000 March	Mid-Atlantic	1274062
2000 March	New England	1132966
2000 March	SouthEast US	1311986
2000 March		3719014
2001 February	Mid-Atlantic	428676
2001 February	New England	615746
2001 February	SouthEast US	618423
2001 February		1662845
2001 January	Mid-Atlantic	610697
2001 January	New England	509215

2001 January	SouthEast US	379021
2001 January		1498933
2001 March	Mid-Atlantic	637031
2001 March	New England	566483
2001 March	SouthEast US	655993
2001 March		1859507
		15063855

25 rows selected.

Запрос преобразует представление месяца из числового в строковое и объединяет его со строковым представлением года. Одно и то же выражение должно использоваться в списке SELECT и в инструкции ROLLUP. Выражение `TO_CHAR(O.YEAR)||' '||TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month')` рассматривается как один составной столбец.

## Каскадная группировка

В Oracle9i один запрос может содержать в инструкции GROUP BY несколько операций ROLLUP, CUBE или GROUPING SETS или их комбинацию. В Oracle8i это невозможно, попытавшись выполнить подобный запрос, вы получите сообщение об ошибке:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY ROLLUP (O.YEAR, O.MONTH), ROLLUP(R.NAME);
GROUP BY ROLLUP (O.YEAR, O.MONTH), ROLLUP(R.NAME)
```

ERROR at line 6:

ORA-30489: Cannot have more than one rollup/cube expression list

Однако тот же запрос работает в Oracle9i:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY ROLLUP (O.YEAR, O.MONTH), ROLLUP(R.NAME);
```

YEAR MONTH	REGION	TOTAL
2000 January	Mid-Atlantic	1221394
2000 January	New England	1018430
2000 January	SouthEast US	758042
2000 January		2997866
2000 February	Mid-Atlantic	857352
2000 February	New England	1231492
2000 February	SouthEast US	1236846
2000 February		3325690

2000 March	Mid-Atlantic	1274062
2000 March	New England	1132966
2000 March	SouthEast US	1311986
2000 March		3719014
2000	Mid-Atlantic	3352808
2000	New England	3382888
2000	SouthEast US	3306874
2000		10042570
2001 January	Mid-Atlantic	610697
2001 January	New England	509215
2001 January	SouthEast US	379021
2001 January		1498933
2001 February	Mid-Atlantic	428676
2001 February	New England	615746
2001 February	SouthEast US	618423
2001 February		1662845
2001 March	Mid-Atlantic	637031
2001 March	New England	566483
2001 March	SouthEast US	655993
2001 March		1859507
2001	Mid-Atlantic	1676404
2001	New England	1691444
2001	SouthEast US	1653437
2001		5021285
	Mid-Atlantic	5029212
	New England	5074332
	SouthEast US	4980311
		15083855

36 rows selected.

Наличие нескольких операций группировки (ROLLUP, CUBE или GROUPING SETS) в инструкции GROUP BY носит название *каскадной группировки (concatenated grouping)*. Результатом каскадной группировки является вывод перекрестного произведения группировок каждой операции. Следовательно, запрос:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY ROLLUP(O.YEAR), ROLLUP(O.MONTH), ROLLUP(R.NAME);
```

ведет себя как CUBE и выводит такой же результат, что и запрос:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY CUBE(O.YEAR, O.MONTH, R.NAME);
```



Так как CUBE содержит обобщения для всех возможных комбинаций столбцов группировки, каскадная группировка CUBE не отличается от обычного CUBE, и все приведенные ниже запросы возвращают тот же результат, что и рассмотренный ранее запрос:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY CUBE (O.YEAR, O.MONTH), CUBE (R.NAME);
```

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY CUBE (O.YEAR), CUBE (O.MONTH, R.NAME);
```

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY CUBE (O.YEAR, O.MONTH), CUBE (O.YEAR, R.NAME);
```

## Каскадные группировки с использованием GROUPING SETS

Каскадные группировки удобно использовать при работе с GROUPING SETS. Так как GROUPING SETS выводит только строки подытогов, можно указать уровни обобщения, которые вы хотели бы видеть в выводе, используя каскадную группировку GROUPING SETS. Каскадная группировка GROUPING SETS (a,b) и GROUPING SETS (c,d) сгенерирует суммарные строки для уровней обобщения (a,c), (a,d), (b,c) и (b,d). Каскадная группировка GROUPING SETS (a,b) и GROUPING SETS (c) выведет суммарные строки для уровней обобщения (a,c) и (b,c). Например:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY GROUPING SETS (O.YEAR, O.MONTH), GROUPING SETS (R.NAME);
```

YEAR MONTH	REGION	TOTAL
2000	Mid-Atlantic	3352808
2000	New England	3382888
2000	SouthEast US	3306874
2001	Mid-Atlantic	1676404
2001	New England	1691444

2001	SouthEast US	1653437
January	Mid-Atlantic	1832091
January	New England	1527645
January	SouthEast US	1137063
February	Mid-Atlantic	1286028
February	New England	1847238
February	SouthEast US	1855269
March	Mid-Atlantic	1911093
March	New England	1699449
March	SouthEast US	1967979

15 rows selected.

**Каскадная группировка** GROUP BY GROUPING SETS (O.YEAR, O.MONTH), GROUPING SETS (R.NAME) порождает строки для уровней обобщения (O.YEAR, R.NAME) и (O.MONTH, R.NAME). Так что в выводе присутствуют суммарные строки для комбинаций (год, регион) и (месяц, регион). Следующий пример расширяет предыдущий запрос:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY GROUPING SETS (O.YEAR, O.MONTH), GROUPING SETS (O.YEAR, R.NAME);
```

1:	YEAR MONTH	REGION	TOTAL
2:	-----		
3:	2000		10042570
4:	2001		5021285
5:	2000 January		2997866
6:	2000 February		3325690
7:	2000 March		3719014
8:	2001 January		1498933
9:	2001 February		1662845
10:	2001 March		1859507
11:	2000	Mid-Atlantic	3352808
12:	2000	New England	3382888
13:	2000	SouthEast US	3306874
14:	2001	Mid-Atlantic	1676404
15:	2001	New England	1691444
16:	2001	SouthEast US	1653437
17:	January	Mid-Atlantic	1832091
18:	January	New England	1527645
19:	January	SouthEast US	1137063
20:	February	Mid-Atlantic	1286028
21:	February	New England	1847238
22:	February	SouthEast US	1855269
23:	March	Mid-Atlantic	1911093
24:	March	New England	1699449
25:	March	SouthEast US	1967979

23 rows selected.



Данный пример выводит четыре комбинации группировок. Различные комбинации группировок, выводимых запросом, и ссылки на соответствующие им номера строк вывода приведены в табл. 12.1.

*Таблица 12.1. Комбинации группировок*

Комбинация группировок	Соответствующие строки
(O.YEAR, O.YEAR)	3–4
(O.YEAR, R.NAME)	11–16
(O.MONTH, O.YEAR)	5–10
(O.MONTH, R.NAME)	17–25

Операция GROUPING SETS не зависит от порядка столбцов, поэтому два предложенных далее запроса выводят те же результаты, что и рассмотренный ранее запрос:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY GROUPING SETS (O.YEAR, R.NAME), GROUPING SETS (O.YEAR, O.MONTH);

SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY GROUPING SETS (O.MONTH, O.YEAR), GROUPING SETS (R.NAME, O.YEAR);
```

Разрешено использование комбинации ROLLUP, CUBE и GROUPING SETS в одной инструкции GROUP BY:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY GROUPING SETS (O.MONTH, O.YEAR), ROLLUP(R.NAME), CUBE (O.YEAR);
```

Однако вывод таких запросов редко оказывается полезным. Прежде чем использовать подобную конструкцию, подумайте, действительно ли она необходима.

## ROLLUP и CUBE как аргументы GROUPING SETS

В отличие от операций ROLLUP и CUBE, операция GROUPING SETS может принимать ROLLUP или CUBE как аргумент. Как вы уже знаете, GROUPING SETS выводит только строки подытогов. Однако иногда требуется вывести наряду с подытогами и общий итог. В таких случаях можно выпол-



нить операцию GROUPING SETS над операцией ROLLUP, как показано в следующем примере:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY GROUPING SETS (ROLLUP (O.YEAR), ROLLUP (O.MONTH), ROLLUP(R. NAME));
```

YEAR MONTH	REGION	TOTAL
-----		
2000		10042570
2001		5021285
	January	4496799
	February	4988535
	March	5578521
	Mid-Atlantic	5029212
	New England	5074332
	SouthEast US	4960311
		15063855
		15063855
		15063855

11 rows selected.

Выводятся подытоги для каждого измерения, как и должно быть при использовании обычной операции GROUPING SETS. Кроме того, выводится общий итог для всех измерений. Однако в выводе присутствуют три одинаковые строки общих итогов. Дело в том, что такую строку порождает каждая из трех операций ROLLUP внутри GROUPING SETS. Если для вас важно наличие ровно одной строки общего итога, можно использовать в инструкции SELECT ключевое слово DISTINCT:

```
SELECT DISTINCT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY GROUPING SETS (ROLLUP (O.YEAR), ROLLUP (O.MONTH), ROLLUP(R. NAME));
```

YEAR MONTH	REGION	TOTAL
-----		
2000		10042570
2001		5021285
	February	4988535
	January	4496799
	March	5578521
	Mid-Atlantic	5029212
	New England	5074332
	SouthEast US	4960311
		15063855

9 rows selected.

Ключевое слово **DISTINCT** уничтожает дубликаты строки общего итога. Можно добиться исключения дубликатов из вывода и при помощи функции **GROUP\_ID**, но об этом чуть позже.

Если вам необходимы подытоги и итоги для составных измерений, используйте в **GROUPING SETS** составные или каскадные операции **ROLLUP**:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total
FROM   ORDERS O, REGION R
WHERE  R.REGION_ID = O.REGION_ID
AND    O.MONTH BETWEEN 1 AND 3
GROUP BY GROUPING SETS (ROLLUP (O.YEAR, O.MONTH), ROLLUP(R.NAME));
```

YEAR MONTH	REGION	TOTAL
2000 January		2997866
2000 February		3325690
2000 March		3719014
2000		10042570
2001 January		1498933
2001 February		1662845
2001 March		1859507
2001		5021285
	Mid-Atlantic	5029212
	New England	5074332
	SouthEast US	4960311
		15063855
		15063855

13 rows selected.

Запрос выводит подытоги для комбинаций (YEAR, MONTH), подытоги для региона, подытоги для года и общий итог. Так как в операции **GROUPING SETS** несколько раз применяется операция **ROLLUP**, в выводе присутствует несколько строк с общим итогом.

## Функции GROUPING\_ID и GROUP\_ID

Ранее в этой главе вы узнали, как использовать функцию **GROUPING** для того, чтобы различать строки обычной инструкции **GROUP BY** и итоговые строки, выведенные расширениями **GROUP BY**. Oracle9i расширяет функциональность **GROUPING** и вводит две новые функции, которые можно использовать вместе с инструкцией **GROUP BY**:

- **GROUPING\_ID**
- **GROUP\_ID**

Эти функции могут использоваться только совместно с инструкцией **GROUP BY**. Но в отличие от функции **GROUPING**, которая может применяться только с расширением **GROUP BY**, функции **GROUPING\_ID** и **GROUP\_ID** могут использоваться в запросе без расширения **GROUP BY**.



Хотя и разрешено применять две вышеназванные функции без расширений GROUP BY, надо сказать, что использование GROUPING\_ID и GROUP\_ID в отсутствие ROLLUP, CUBE или GROUPING SETS приводит к получению малоинформативного результата, так как для всех обычных строк GROUP BY функции GROUPING\_ID и GROUP\_ID равны 0.

В последующих разделах две эти функции будут описаны подробно.

## GROUPING\_ID

Синтаксис функции GROUPING ID таков:

```
SELECT ... , GROUPING_ID(упорядоченный список столбцов группировки)
FROM ...
GROUP BY ...
```

Функция GROUPING\_ID получает в качестве аргумента упорядоченный список столбцов группировки и вычисляет результат, выполняя следующие шаги:

1. Сначала функция GROUPING\_ID генерирует результаты функции GROUPING для каждого отдельного столбца списка. Результатом выполнения этого шага является набор нулей и единиц.
2. Затем она размещает единицы и нули в порядке, повторяющем порядок столбцов в списке аргументов, образуя битовый вектор.
3. Рассматривая этот битовый вектор (последовательность нулей и единиц) как двоичное число, она преобразует его в десятичное число.
4. Полученное в шаге 3 десятичное число возвращается в качестве результата функции GROUPING\_ID.

Следующий пример иллюстрирует данный процесс и сравнивает результаты функции GROUPING\_ID с результатами GROUPING:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total,
       GROUPING(O.YEAR) Y, GROUPING(O.MONTH) M, GROUPING(R.NAME) R,
       GROUPING_ID (O.YEAR, O.MONTH, R.NAME) GID
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY CUBE (O.YEAR, O.MONTH, R.NAME);
```

YEAR MONTH	REGION	TOTAL	Y	M	R	GID
2000 January	Mid-Atlantic	1221394	0	0	0	0
2000 January	New England	1018430	0	0	0	0
2000 January	SouthEast US	758042	0	0	0	0
2000 January		2997866	0	0	1	1
2000 February	Mid-Atlantic	857352	0	0	0	0



2000	February	New England	1231492	0	0	0	0
2000	February	SouthEast US	1236846	0	0	0	0
2000	February		3325690	0	0	1	1
2000	March	Mid-Atlantic	1274062	0	0	0	0
2000	March	New England	1132966	0	0	0	0
2000	March	SouthEast US	1311986	0	0	0	0
2000	March		3719014	0	0	1	1
2000		Mid-Atlantic	3352808	0	1	0	2
2000		New England	3382888	0	1	0	2
2000		SouthEast US	3306874	0	1	0	2
2000			10042570	0	1	1	3
2001	January	Mid-Atlantic	610697	0	0	0	0
2001	January	New England	509215	0	0	0	0
2001	January	SouthEast US	379021	0	0	0	0
2001	January		1498933	0	0	1	1
2001	February	Mid-Atlantic	428676	0	0	0	0
2001	February	New England	615746	0	0	0	0
2001	February	SouthEast US	618423	0	0	0	0
2001	February		1862845	0	0	1	1
2001	March	Mid-Atlantic	637031	0	0	0	0
2001	March	New England	566483	0	0	0	0
2001	March	SouthEast US	655993	0	0	0	0
2001	March		1859507	0	0	1	1
2001		Mid-Atlantic	1676404	0	1	0	2
2001		New England	1691444	0	1	0	2
2001		SouthEast US	1653437	0	1	0	2
2001			5021285	0	1	1	3
	January	Mid-Atlantic	1832091	1	0	0	4
	January	New England	1527645	1	0	0	4
	January	SouthEast US	1137063	1	0	0	4
	January		4496799	1	0	1	5
	February	Mid-Atlantic	1286028	1	0	0	4
	February	New England	1847238	1	0	0	4
	February	SouthEast US	1855269	1	0	0	4
	February		4988535	1	0	1	5
	March	Mid-Atlantic	1911093	1	0	0	4
	March	New England	1699449	1	0	0	4
	March	SouthEast US	1987979	1	0	0	4
	March		5578521	1	0	1	5
		Mid-Atlantic	5029212	1	1	0	6
		New England	5074332	1	1	0	6
		SouthEast US	4960311	1	1	0	6
			15063855	1	1	1	7

48 rows selected.

Обратите внимание, что результат функции GROUPING\_ID является десятичным эквивалентом битового вектора, порожденного отдельными функциями GROUPING. В данном выводе GROUPING\_ID имеет значения 0, 1, 2, 3, 4, 5, 6 и 7. Эти уровни обобщения описаны в табл. 12.2.

Таблица 12.2. Результат GROUPING\_ID(O.YEAR, O.MONTH, R.NAME)

Уровень агрегации	Битовый вектор	GROUPING_ID
Обычные строки GROUP BY	0 0 0	0
Подытог для года-месяца, обобщение по региону	0 0 1	1
Подытог для года-региона, обобщение по месяцу	0 1 0	2
Подытог для года, обобщение по месяцу-региону	0 1 1	3
Подытог для месяца-региона, обобщение по году	1 0 0	4
Подытог для месяца, обобщение по году-региону	1 0 1	5
Подытог для региона, обобщение по году-месяцу	1 1 0	6
Общий итог для всех уровней, обобщение по году-месяцу-региону	1 1 1	7

Функцию GROUPING\_ID можно использовать в запросе для эффективной фильтрации строк согласно вашим требованиям. Предположим, что вы хотите вывести только итоговые строки и не выводить обычные строки GROUP BY. Можно использовать функцию GROUPING\_ID в инструкции HAVING, чтобы ограничить вывод только теми строками, которые содержат подытоги и итоги (теми, для которых GROUPING\_ID > 0):

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY CUBE (O.YEAR, O.MONTH, R.NAME)
HAVING GROUPING_ID (O.YEAR, O.MONTH, R.NAME) > 0;
```

YEAR	MONTH	REGION	TOTAL
2000	January		2997866
2000	February		3325690
2000	March		3719014
2000		Mid-Atlantic	3352808
2000		New England	3382888
2000		SouthEast US	3308874
2000			10042570
2001	January		1498933
2001	February		1662845
2001	March		1859507
2001		Mid-Atlantic	1676404
2001		New England	1691444
2001		SouthEast US	1653437
2001			5021285
	January	Mid-Atlantic	1832091
	January	New England	1527645

January	SouthEast US	1137063
January		4496799
February	Mid-Atlantic	1286028
February	New England	1847238
February	SouthEast US	1855269
February		4988535
March	Mid-Atlantic	1911093
March	New England	1699449
March	SouthEast US	1967979
March		5578521
	Mid-Atlantic	5029212
	New England	5074332
	SouthEast US	4960311
		15063855

30 rows selected.

Как видите, функция GROUPING\_ID упрощает фильтрацию вывода обобщающих операций. Если бы такой функции не было, пришлось бы писать сложный запрос, применяя для получения того же результата функцию GROUPING. Следующий запрос для вывода только итогов и подытогов использует GROUPING вместо функции GROUPING\_ID. Заметьте, как усложнилась инструкция HAVING.

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
R.NAME REGION, SUM(O.TOT_SALES) Total
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY CUBE (O.YEAR, O.MONTH, R.NAME)
HAVING GROUPING(O.YEAR) > 0
OR GROUPING(O.MONTH) > 0
OR GROUPING(R.NAME) > 0;
```

YEAR	MONTH	REGION	TOTAL
2000	January		2997866
2000	February		3325690
2000	March		3719014
2000		Mid-Atlantic	3352808
2000		New England	3382888
2000		SouthEast US	3306874
2000			10042570
2001	January		1498933
2001	February		1662845
2001	March		1859507
2001		Mid-Atlantic	1676404
2001		New England	1691444
2001		SouthEast US	1653437
2001			5021285
	January	Mid-Atlantic	1832091
	January	New England	1527645



January	SouthEast US	1137063
January		4496799
February	Mid-Atlantic	1286028
February	New England	1847238
February	SouthEast US	1855269
February		4988535
March	Mid-Atlantic	1911093
March	New England	1699449
March	SouthEast US	1967979
March		5578521
	Mid-Atlantic	5029212
	New England	5074332
	SouthEast US	4960311
		15063855

30 rows selected.

## GROUP\_ID

В предыдущем разделе было показано, что Oracle9i разрешает повторять в инструкции GROUP BY названия столбцов и использовать в ней несколько группирующих операций. Некоторые комбинации приводят к появлению в выходных данных строк-дубликатов. Функция GROUP\_ID позволяет выявлять повторяющиеся строки.

Синтаксис функции GROUP\_ID таков:

```
SELECT ... , GROUP_ID()
FROM ...
GROUP BY ...
```

Функция GROUP\_ID не получает аргументов и возвращает число от 0 до (n-1), где n – счетчик количества дубликатов. Первое вхождение заданной строки в выводе запроса будет иметь значение GROUP\_ID, равное 0, второе вхождение этой же строки будет иметь значение GROUP\_ID, равное 1, и т. д. Следующий пример иллюстрирует использование функции GROUP\_ID:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total, GROUP_ID()
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY O.YEAR, ROLLUP (O.YEAR, O.MONTH, R.NAME);
```

YEAR	MONTH	REGION	TOTAL	GROUP_ID()
2000	January	Mid-Atlantic	1221394	0
2000	January	New England	1018430	0
2000	January	SouthEast US	758042	0
2000	January		2997866	0
2000	February	Mid-Atlantic	857352	0

2000	February	New England	1231492	0
2000	February	SouthEast US	1236846	0
2000	February		3325690	0
2000	March	Mid-Atlantic	1274062	0
2000	March	New England	1132966	0
2000	March	SouthEast US	1311986	0
2000	March		3719014	0
2001	January	Mid-Atlantic	610697	0
2001	January	New England	509215	0
2001	January	SouthEast US	379021	0
2001	January		1498933	0
2001	February	Mid-Atlantic	428676	0
2001	February	New England	615746	0
2001	February	SouthEast US	618423	0
2001	February		1662845	0
2001	March	Mid-Atlantic	637031	0
2001	March	New England	566483	0
2001	March	SouthEast US	655993	0
2001	March		1859507	0
2000			10042570	0
2001			5021285	0
2000			10042570	1
2001			5021285	1

28 rows selected.

Обратите внимание на значение 1, возвращенное GROUP\_ID для двух последних строк. Эти строки действительно являются дубликатами двух предыдущих строк. Если вы не хотите, чтобы в выводе присутствовали повторения, ограничьте результаты запроса строками, для которых значение GROUP\_ID равно 0:

```
SELECT O.YEAR, TO_CHAR(TO_DATE(O.MONTH, 'MM'), 'Month') MONTH,
       R.NAME REGION, SUM(O.TOT_SALES) Total, GROUP_ID()
FROM ORDERS O, REGION R
WHERE R.REGION_ID = O.REGION_ID
AND O.MONTH BETWEEN 1 AND 3
GROUP BY O.YEAR, ROLLUP (O.YEAR, O.MONTH, R.NAME)
HAVING GROUP_ID() = 0;
```

YEAR	MONTH	REGION	TOTAL	GROUP_ID()
2000	January	Mid-Atlantic	1221394	0
2000	January	New England	1018430	0
2000	January	SouthEast US	758042	0
2000	January		2997866	0
2000	February	Mid-Atlantic	857352	0
2000	February	New England	1231492	0
2000	February	SouthEast US	1236846	0
2000	February		3325690	0
2000	March	Mid-Atlantic	1274062	0
2000	March	New England	1132966	0

2000 March	SouthEast US	1311986	0
2000 March		3719014	0
2001 January	Mid-Atlantic	610697	0
2001 January	New England	509215	0
2001 January	SouthEast US	379021	0
2001 January		1498933	0
2001 February	Mid-Atlantic	428676	0
2001 February	New England	615746	0
2001 February	SouthEast US	618423	0
2001 February		1662845	0
2001 March	Mid-Atlantic	637031	0
2001 March	New England	566483	0
2001 March	SouthEast US	655993	0
2001 March		1859507	0
2000		10042570	0
2001		5021285	0

26 rows selected.

Данная версия запроса использует инструкцию `HAVING GROUP_ID = 0` для удаления из результирующего множества двух дубликатов итогов. Функция `GROUP_ID` имеет смысл только в инструкции `HAVING`, так как применяется к итоговому данным. Использовать `GROUP_ID` в инструкции `WHERE` нельзя, и даже пытаться не стоит.



# 13

## Аналитический SQL

Годами SQL критиковали за неспособность обрабатывать запросы, характерные для систем поддержки принятия решений. Благодаря множеству новых аналитических функций, введенных в Oracle8i и Oracle9i, сделан огромный шаг по устранению этого недостатка. Тем самым Oracle сделал еще более расплывчатой границу между своим многоцелевым сервером реляционных баз данных и другими специализированными хранилищами данных и серверами статистического анализа.

### Обзор аналитического SQL

Запросы систем поддержки принятия решений (DSS, Decision Support Systems) отличаются от запросов в системах оперативной обработки транзакций (OLTP, Online Transaction Processing). Рассмотрим следующие бизнес-запросы:

- Найти десять лучших продавцов в каждом районе за прошлый год.
- Найти всех клиентов, годовой объем заказов которых превысил 20% от общего объема продаж их географического региона.
- Определить, какой регион больше всего пострадал от поквартального спада продаж.
- Найти товары, продаваемые лучше и хуже других для каждого квартала прошедшего года по штатам.

Подобные запросы являются основой систем DSS и используются руководителями, аналитиками, маркетологами и т. д. для отслеживания тенденций, выявления отклонений, неиспользованных возможностей и предсказания дальнейшей эффективности работы. Системы DSS обычно работают с хранилищами данных, содержащими огромное ко-

личество обобщенных данных, предоставляющих плодородную почву для исследований и формулирования бизнес-решений.

Хотя все упомянутые запросы легко сформулировать на разговорном языке, записать их на SQL раньше было сложно по следующим причинам:

- Они могут требовать различные уровни обобщения для одних и тех же данных.
- Они могут включать сравнения данных внутри одной таблицы (например, сравнение одной или нескольких строк таблицы с другими строками той же таблицы).
- Они могут требовать дополнительной фильтрации после проведения сортировки результирующего множества (нахождение десяти лучших и десяти худших продавцов прошлого месяца).

Можно было бы получить желаемые результаты, используя такие возможности SQL, как самообъединения, встроенные представления и пользовательские функции, но полученные таким способом запросы было бы тяжело понять, а выполнялись бы они неприемлемо долго. Чтобы пояснить сложность написания таких запросов, попробуем построить запрос: «Найти всех клиентов, годовой объем заказов которых превысил 20% от общего объема продаж их географического региона».

В этом и всех других примерах данной главы будем использовать простую схему, состоящую из одной реальной таблицы `orders`, содержащей обобщенную информацию о продажах по следующим измерениям: регион, продавец, клиент и месяц. У этого запроса два аспекта, каждый из которых требует своего уровня обобщения одних и тех же данных:

1. Обобщить все продажи прошлого года по регионам.
2. Обобщить все продажи прошлого года по клиентам.

После получения двух таких промежуточных результатов необходимо сравнить итог каждого клиента с итогом его региона и посмотреть, превышает ли он 20%. В конечном результирующем множестве будет присутствовать фамилия клиента, общий объем его заказов, процент от объема продаж региона и название региона.

Запрос, обобщающий объемы продаж по регионам, выглядит так:

```
SELECT o.region_id region_id, SUM(o.tot_sales) tot_sales
FROM orders o
WHERE o.year = 2001
GROUP BY o.region_id;
```

REGION_ID	TOT_SALES
-----------	-----------

5	6585641
6	6307766
7	6868495
8	6853015
9	6739374
10	6238901

Теперь напомним запрос, обобщающий объемы продаж по клиентам:

```
SELECT o.cust_nbr cust_nbr, o.region_id region_id,
       SUM(o.tot_sales) tot_sales
FROM orders o
WHERE o.year = 2001
GROUP BY o.cust_nbr, o.region_id;
```

CUST_NBR	REGION_ID	TOT_SALES
1	5	1151162
2	5	1224992
3	5	1161286
4	5	1878275
5	5	1169926
6	6	1788836
7	6	971585
8	6	1141638
9	6	1208959
10	6	1196748
11	7	1190421
12	7	1182275
13	7	1310434
14	7	1929774
15	7	1255591
16	8	1068467
17	8	1944281
18	8	1253840
19	8	1174421
20	8	1412006
21	9	1020541
22	9	1036146
23	9	1224992
24	9	1224992
25	9	2232703
26	10	1808949
27	10	1322747
28	10	986964
29	10	903383
30	10	1216858

Поместив каждый из запросов во встроенное представление и объединив их по столбцу `region_id`, можно найти клиентов, для которых объем заказов превышает 20% от объема продаж их региона:

```
SELECT cust_sales.cust_nbr cust_nbr, cust_sales.region_id region_id,
       cust_sales.tot_sales cust_sales, region_sales.tot_sales region_sales
FROM
  (SELECT o.region_id region_id, SUM(o.tot_sales) tot_sales
   FROM orders o
   WHERE o.year = 2001
   GROUP BY o.region_id) region_sales,
  (SELECT o.cust_nbr cust_nbr, o.region_id region_id,
```



```

    SUM(o.tot_sales) tot_sales
FROM orders o
WHERE o.year = 2001
GROUP BY o.cust_nbr, o.region_id) cust_sales
WHERE cust_sales.region_id = region_sales.region_id
AND cust_sales.tot_sales > (region_sales.tot_sales * .2);

```

CUST_NBR	REGION_ID	CUST_SALES	REGION_SALES
4	5	1878275	6585641
6	6	1788836	6307766
14	7	1929774	6868495
17	8	1944281	6853015
20	8	1412006	6853015
25	9	2232703	6739374
26	10	1808949	6238901
27	10	1322747	6238901

На последнем этапе объединим клиентское измерение с региональным, чтобы включить в результирующее множество фамилию заказчика и название региона:

```

SELECT c.name cust_name,
       big_custs.cust_sales cust_sales, r.name region_name,
       100 * ROUND(big_custs.cust_sales /
                   big_custs.region_sales, 2) percent_of_region
FROM region r, customer c,
     (SELECT cust_sales.cust_nbr cust_nbr, cust_sales.region_id region_id,
          cust_sales.tot_sales cust_sales,
          region_sales.tot_sales region_sales
      FROM
        (SELECT o.region_id region_id, SUM(o.tot_sales) tot_sales
         FROM orders o
         WHERE o.year = 2001
         GROUP BY o.region_id) region_sales,
        (SELECT o.cust_nbr cust_nbr, o.region_id region_id,
             SUM(o.tot_sales) tot_sales
         FROM orders o
         WHERE o.year = 2001
         GROUP BY o.cust_nbr, o.region_id) cust_sales
      WHERE cust_sales.region_id = region_sales.region_id
      AND cust_sales.tot_sales > (region_sales.tot_sales * .2)) big_custs
WHERE big_custs.cust_nbr = c.cust_nbr
AND big_custs.region_id = r.region_id;

```

CUST_NAME	CUST_SALES	REGION_NAME	PERCENT_OF_REGION
Flowtech Inc.	1878275	New England	29
Spartan Industries	1788836	Mid-Atlantic	28
Madden Industries	1929774	SouthEast US	28
Evans Supply Corp.	1944281	SouthWest US	28
Malden Labs	1412006	SouthWest US	21

Worcester Technologies	2232703 NorthWest US	33
Alpha Technologies	1808949 Central US	29
Phillips Labs	1322747 Central US	21

Используя встроенные представления (никакой экзотики!), мы построили запрос, который в одиночку порождает желаемый результат. Однако такое решение имеет следующие недостатки:

- Запрос достаточно сложен.
- Для формирования различных уровней обобщения, необходимых запросу, требуется два обхода одних и тех же строк таблицы заказов.

Давайте посмотрим, как можно упростить запрос и выполнить ту же работу за один проход таблицы заказов, используя одну из новых аналитических функций. Вместо того чтобы создавать два разных запроса для обобщения данных по региону и по клиенту, напомним единый запрос, который будет обобщать информацию и по клиенту, и по региону. Затем можно будет вызвать аналитическую функцию, которая выполнит обобщение следующего уровня для получения общего объема продаж для региона:

```
1 SELECT o.region_id region_id, o.cust_nbr cust_nbr,
2       SUM(o.tot_sales) tot_sales,
3       SUM(SUM(o.tot_sales)) OVER (PARTITION BY o.region_id) region_sales
4 FROM orders o
5 WHERE o.year = 2001
6 GROUP BY o.region_id, o.cust_nbr;
REGION_ID  CUST_NBR  TOT_SALES  REGION_SALES
```

5	1	1151162	6584167
5	2	1223518	6584167
5	3	1161286	6584167
5	4	1878275	6584167
5	5	1169926	6584167
6	6	1788836	6307766
6	7	971585	6307766
6	8	1141638	6307766
6	9	1208959	6307766
6	10	1196748	6307766
7	11	1190421	6868495
7	12	1182275	6868495
7	13	1310434	6868495
7	14	1929774	6868495
7	15	1255591	6868495
8	16	1068467	6853015
8	17	1944281	6853015
8	18	1253840	6853015
8	19	1174421	6853015
8	20	1412006	6853015
9	21	1020541	6726929
9	22	1036146	6726929
9	23	1212547	6726929



9	24	1224992	6726929
9	25	2232703	6726929
10	26	1808949	6238901
10	27	1322747	6238901
10	28	986964	6238901
10	29	903383	6238901
10	30	1216858	6238901

Аналитическая функция присутствует в строке 3 предыдущего запроса, а результат имеет псевдоним *region\_sales*. Обобщающая функция ( $\text{SUM}(o.\text{tot\_sales})$ ) в строке 2 генерирует суммарный объем продаж по клиенту и региону, как предписывает инструкция *GROUP BY*, а аналитическая функция в строке 3 обобщает эти суммы для каждого региона, вычисляя объем продаж региона. Значение столбца *region\_sales* одинаково для всех клиентов одного региона и равно сумме заказов всех клиентов данного региона. Затем запрос помещается во встроенное представление<sup>1</sup>, клиенты, объем заказов которых меньше 20% от объема продаж их региона, отфильтровываются и таблицы регионов и клиентов объединяются для получения нужного результата:

```
SELECT c.name cust_name,
       cust_sales.tot_sales cust_sales, r.name region_name,
       100 * ROUND(cust_sales.tot_sales /
                   cust_sales.region_sales, 2) percent_of_region
FROM region r, customer c,
     (SELECT o.region_id region_id, o.cust_nbr cust_nbr,
          SUM(o.tot_sales) tot_sales,
          SUM(SUM(o.tot_sales)) OVER (PARTITION BY o.region_id) region_sales
     FROM orders o
     WHERE o.year = 2001
     GROUP BY o.region_id, o.cust_nbr) cust_sales
WHERE cust_sales.tot_sales > (cust_sales.region_sales * .2)
   AND cust_sales.region_id = r.region_id
   AND cust_sales.cust_nbr = c.cust_nbr;
```

CUST_NAME	CUST_SALES	REGION_NAME	PERCENT_OF_REGION
Flowtech Inc.	1878275	New England	29
Spartan Industries	1788836	Mid-Atlantic	28
Madden Industries	1929774	SouthEast US	28
Evans Supply Corp.	1944281	SouthWest US	28
Malden Labs	1412006	SouthWest US	21
Worcester Technologies	2232703	NorthWest US	33
Alpha Technologies	1808949	Central US	29
Phillips Labs	1322747	Central US	21

<sup>1</sup> Использование встроенного представления избавляет от необходимости объединять таблицы регионов и заказчиков с таблицей заказов. В противном случае пришлось бы включать столбцы таблиц клиентов и регионов в инструкцию *GROUP BY*.



Не вдаваясь в детали работы функции `SUM...OVER` (вернемся к этому в разделе «Функции для создания отчетов»), можно понять, что Oracle выполняет обобщение обобщения вместо того, чтобы дважды просматривать строки заказов. Запрос станет более быстрым и должен оказаться более простым для понимания и сопровождения, как только вы освоите его синтаксис.

В отличие от встроенных функций, таких как `DECODE`, `GREATEST` и `SUBSTR`, аналитические функции Oracle могут использоваться только в инструкции `SELECT` запроса. Дело в том, что аналитические функции выполняются только *после* того, как вычислены инструкции `FROM`, `WHERE`, `GROUP BY` и `HAVING`. После выполнения аналитических функций вычисляется инструкция `ORDER BY`, упорядочивающая итоговое результирующее множество. Инструкции `ORDER BY` разрешено ссылаться на столбцы инструкции `SELECT`, которые получены с помощью аналитических функций.

В оставшейся части главы будут представлены аналитические функции Oracle8i и Oracle9i, сгруппированные по их функциональности.

## Ранжирующие функции

Определение эффективности некоторой бизнес-сущности по сравнению с такими же, как она (одноранговыми), объектами является основной множества задач принятия решений, в том числе:

- Определение наиболее используемых активов.
- Определение продуктов, хуже всего продающихся в регионе.
- Нахождение лучших торговых представителей.

До выхода Oracle8i программисты могли использовать инструкцию `ORDER BY` для сортировки результирующего множества по одному или нескольким столбцам, но последующая обработка с целью вычисления рангов или перцентилей должна была выполняться с помощью процедурного языка. Однако в Oracle8i разработчики получают возможность использовать ряд новых функций как для формирования рангов для каждой строки, так и для группировки строк с целью выполнения статистических расчетов.

## RANK, DENSE\_RANK и ROW\_NUMBER

Функции `RANK`, `DENSE_RANK` и `ROW_NUMBER` генерируют целое значение от 1 до N для каждой строки, где N меньше или равно количеству строк результирующего множества. Отличие значений, возвращаемых этими функциями, обусловлено тем, как каждая из них обрабатывает равные значения:

- `ROW_NUMBER` возвращает уникальное число для каждой строки, начиная с 1. Для строк, имеющих повторяющиеся значения, числа присваиваются произвольным образом.

- DENSE\_RANK присваивает уникальный номер каждой строке, начиная с 1, за исключением строк, имеющих одинаковые значения; таким строкам присваиваются одинаковые ранги.
- RANK присваивает уникальный номер каждой строке, начиная с 1, за исключением строк, имеющих одинаковые значения; таким строкам присваиваются одинаковые ранги, и в последовательности присваиваемых значений возникает промежуток.

Чтобы пояснить вышесказанное, сформируем классификацию клиентов на основе их годовых закупок. Напишем запрос для вывода информации о продажах за 2001 год:

```
SELECT region_id, cust_nbr, SUM(tot_sales) cust_sales
FROM orders
WHERE year = 2001
GROUP BY region_id, cust_nbr
ORDER BY region_id, cust_nbr;
```

REGION_ID	CUST_NBR	CUST_SALES
5	1	1151162
5	2	1224992
5	3	1161286
5	4	1878275
5	5	1169926
6	6	1788836
6	7	971585
6	8	1141638
6	9	1208959
6	10	1196748
7	11	1190421
7	12	1182275
7	13	1310434
7	14	1929774
7	15	1255591
8	16	1088487
8	17	1944281
8	18	1253840
8	19	1174421
8	20	1412006
9	21	1020541
9	22	1036146
9	23	1224992
9	24	1224992
9	25	2232703
10	26	1808949
10	27	1322747
10	28	986964
10	29	903383
10	30	1216858



Обратите внимание, что три клиента (2, 23 и 24) имеют одинаковые значения общего объема продаж (\$1 224 992). В следующем запросе добавим вызовы трех функций для формирования ранга каждого клиента по всем регионам и упорядочим результирующее множество с помощью функции ROW\_NUMBER, чтобы сделать различия в классификации более заметными:

```
SELECT region_id, cust_nbr,
       SUM(tot_sales) cust_sales,
       RANK() OVER (ORDER BY SUM(tot_sales) DESC) sales_rank,
       DENSE_RANK() OVER (ORDER BY SUM(tot_sales) DESC) sales_dense_rank,
       ROW_NUMBER() OVER (ORDER BY SUM(tot_sales) DESC) sales_number
FROM orders
WHERE year = 2001
GROUP BY region_id, cust_nbr
ORDER BY 6;
```

REGION_ID	CUST_NBR	CUST_SALES	SALES_RANK	SALES_DENSE_RANK	SALES_NUMBER
9	25	2232703	1	1	1
8	17	1944281	2	2	2
7	14	1929774	3	3	3
5	4	1878275	4	4	4
10	26	1808949	5	5	5
6	6	1788836	6	6	6
8	20	1412006	7	7	7
10	27	1322747	8	8	8
7	13	1310434	9	9	9
7	15	1255591	10	10	10
8	18	1253840	11	11	11
5	2	1224992	12	12	12
9	23	1224992	12	12	13
9	24	1224992	12	12	14
10	30	1216858	15	13	15
6	9	1208959	16	14	16
6	10	1196748	17	15	17
7	11	1190421	18	16	18
7	12	1182275	19	17	19
8	19	1174421	20	18	20
5	5	1169926	21	19	21
5	3	1161286	22	20	22
5	1	1151162	23	21	23
6	8	1141638	24	22	24
8	16	1068467	25	23	25
9	22	1036146	26	24	26
9	21	1020541	27	25	27
10	28	986964	28	26	28
6	7	971585	29	27	29
10	29	903383	30	28	30

Пусть вас не смущает инструкция ORDER BY в конце запроса и инструкции ORDER BY в вызове каждой функции; функции используют свою ин-



струкцию ORDER BY для внутренних целей – сортировки результатов в целях присвоения ранга. То есть каждая из трех функций применяет свой алгоритм ранжирования к сумме продаж каждого клиента в порядке убывания. Последняя инструкция ORDER BY указывает в качестве ключа сортировки итогового результирующего множества результаты функции ROW\_NUMBER, но можно было выбрать ключом сортировки любой из шести столбцов.

Функции RANK и DENSE\_RANK присваивают ранг 12 трем строкам с общим объемом продаж \$1 224 992, в то время как функция ROW\_NUMBER присваивает тем же трем строкам ранги 12, 13 и 14. Различие функций RANK и DENSE\_RANK проявляется в том, какой ранг они присваивают следующей строке с более низким объемом продаж. Функция RANK оставляет пробел в последовательности рангов и присваивает клиенту с номером 30 ранг 15, в то время как функция DENSE\_RANK продолжает непрерывную последовательность, присваивая ранг 13.

Выбор конкретной функции для использования зависит от желаемого результата. Если бы требовалось выделить из результирующего множества 13 лучших клиентов, следовало бы использовать:

- ROW\_NUMBER, если бы вас интересовало ровно 13 строк независимо от наличия равных значений. В таком случае один из заказчиков, который мог бы попасть в список лучших, будет исключен из результирующего множества.
- RANK, если вы хотите получить как минимум 13 строк, но не хотите включать в список строки, которые бы в него не попали, если бы не было этого ограничения. В этом случае будет извлечено 14 строк.
- DENSE\_RANK, если вас интересуют все клиенты с рангом 13 или меньше, включая все повторения. Будет извлечено 15 строк.

Предыдущий запрос выводит ранги для всего результирующего множества, но можно выводить и независимые наборы рангов для нескольких разделов результата. Следующий запрос генерирует ранги по продажам для клиентов внутри каждого региона, а не для всех регионов сразу. В запрос добавлена инструкция PARTITION BY:

```
SELECT region_id, cust_nbr, SUM(tot_sales) cust_sales,
       RANK() OVER (PARTITION BY region_id
                    ORDER BY SUM(tot_sales) DESC) sales_rank,
       DENSE_RANK() OVER (PARTITION BY region_id
                           ORDER BY SUM(tot_sales) DESC) sales_dense_rank,
       ROW_NUMBER() OVER (PARTITION BY region_id
                           ORDER BY SUM(tot_sales) DESC) sales_number
FROM orders
WHERE year = 2001
GROUP BY region_id, cust_nbr
ORDER BY 1,6;
```

REGION_ID	CUST_NBR	CUST_SALES	SALES_RANK	SALES_DENSE_RANK	SALES_NUMBER
5	4	1878275	1	1	1

5	2	1224992	2	2	2
5	5	1169926	3	3	3
5	3	1161286	4	4	4
5	1	1151162	5	5	5
6	6	1788536	1	1	1
6	9	1208959	2	2	2
6	10	1196748	3	3	3
8	8	1141638	4	4	4
6	7	971585	5	5	5
7	14	1929774	1	1	1
7	13	1310434	2	2	2
7	15	1255591	3	3	3
7	11	1190421	4	4	4
7	12	1182275	5	5	5
8	17	1944281	1	1	1
8	20	1412006	2	2	2
8	18	1253840	3	3	3
8	19	1174421	4	4	4
8	16	1068487	5	5	5
9	25	2232703	1	1	1
9	23	1224992	2	2	2
9	24	1224992	2	2	3
9	22	1036146	4	3	4
9	21	1020541	5	4	5
10	26	1808949	1	1	1
10	27	1322747	2	2	2
10	30	1216858	3	3	3
10	28	986964	4	4	4
10	29	903383	5	5	5

Каждый клиент получает ранг от 1 до 5 в зависимости от положения относительно других клиентов того же региона. Двое из трех клиентов с одинаковыми объемами заказов относятся к региону с номером 9, и, как и раньше, функции RANK и DENSE\_RANK присваивают двум таким клиентам одинаковые ранги.



Инструкция PARTITION BY используется в ранжирующих функциях для того, чтобы разделить результирующее множество на части, к которым будет применено независимое ранжирование. Она абсолютно не похожа на применяемые для разбиения на части таблицы или индекса инструкции PARTITION BY RANGE/HASH/LIST, описанные в главе 10.

## Обработка значений NULL

Все ранжирующие функции позволяют при вызове указать, где именно в классификации должны появляться значения NULL. Для этого в конец инструкции ORDER BY функции добавляется NULLS FIRST или NULLS LAST:

```
SELECT region_id, cust_nbr, SUM(tot_sales) cust_sales,
       RANK() OVER (ORDER BY SUM(tot_sales) DESC NULLS LAST) sales_rank
FROM orders
```



```
WHERE year = 2001
GROUP BY region_id, cust_nbr;
```

Если ничего не указано, значения NULL будут последними в нисходящих классификациях и первыми – в восходящих.

## Запросы N лучших/худших

Ранжированное множество часто применяется для определения N лучших или худших исполнителей. Поскольку вызывать аналитические функции из инструкций WHERE и HAVING нельзя, приходится формировать ранги для всех строк, а затем использовать внешний запрос для отбрасывания ненужных. Например, следующий запрос использует встроенное представление для определения пяти лучших продавцов 2001 года:

```
SELECT s.name, sp.sp_sales total_sales
FROM salesperson s,
     (SELECT salesperson_id, SUM(tot_sales) sp_sales,
          RANK() OVER (ORDER BY SUM(tot_sales) DESC) sales_rank
      FROM orders
      WHERE year = 2001
      GROUP BY salesperson_id) sp
WHERE sp.sales_rank <= 5
      AND sp.salesperson_id = s.salesperson_id
ORDER BY sp.sales_rank;
```

NAME	TOTAL_SALES
Jeff Blake	1927580
Sam Houseman	1814327
Mark Russell	1784596
John Boorman	1768813
Carl Isaacs	1761814

## Первый/последний

Функции, возвращающей только N лучших или худших строк ранжированного результирующего множества, не существует, но поддерживается функциональность, позволяющая определить первую (самую лучшую) или последнюю (самую худшую) запись ранжированного множества. Такая возможность полезна для запросов типа: «Найти регионы с лучшим и худшим объемом продаж за прошлый год». В отличие от предыдущего запроса о пяти лучших продавцах, этому запросу для ответа на вопрос требуется дополнительная информация – размер результирующего множества.

Oracle9i предоставляет возможность эффективно выполнять подобные запросы, используя функцию, ранжирующую результирующее множество на основе заданного порядка, определяющую строку с наибольшим или наименьшим рангом и выводящую отчет по любому столбцу результирующего множества. Такие функции состоят из трех частей:



1. Инструкции ORDER BY, которая указывает, как ранжировать результирующее множество.
2. Ключевых слов FIRST или LAST, указывающих, какая из строк требуется: с наибольшим или наименьшим рангом.
3. Обобщающей функции (то есть MIN, MAX, AVG, COUNT), которая используется для разрешения конфликтов в случае, если несколько строк результирующего множества имеют одинаковый наибольший или наименьший ранг.

В следующем запросе для нахождения регионов, которые ранжированы как первый и последний по общему объему продаж, используется обобщающая функция MIN:

```
SELECT
  MIN(region_id)
    KEEP (DENSE_RANK FIRST ORDER BY SUM(tot_sales) DESC) best_region,
  MIN(region_id)
    KEEP (DENSE_RANK LAST ORDER BY SUM(tot_sales) DESC) worst_region
FROM orders
WHERE year = 2001
GROUP BY region_id;

BEST_REGION WORST_REGION
-----
              7          10
```

Использование функции MIN может сбивать с толку: она применяется, только если несколько регионов занимают первое или последнее место в классификации. Если бы такая ситуация имела место, была бы выбрана строка с наименьшим значением region\_id. Чтобы узнать, имеются ли на самом деле совпадения, можно дважды вызвать каждую функцию, первый раз используя MIN, а второй — MAX, и посмотреть, одинаковые ли результаты будут возвращены:

```
SELECT
  MIN(region_id)
    KEEP (DENSE_RANK FIRST ORDER BY SUM(tot_sales) DESC) min_best_region,
  MAX(region_id)
    KEEP (DENSE_RANK FIRST ORDER BY SUM(tot_sales) DESC) max_best_region,
  MIN(region_id)
    KEEP (DENSE_RANK LAST ORDER BY SUM(tot_sales) DESC) min_worst_region,
  MAX(region_id)
    KEEP (DENSE_RANK LAST ORDER BY SUM(tot_sales) DESC) max_worst_region
FROM orders
WHERE year = 2001
GROUP BY region_id;

MIN_BEST_REGION MAX_BEST_REGION MIN_WORST_REGION MAX_WORST_REGION
-----
              7              7              10              10
```

В данном случае ни первый, ни последний результат не повторяются. Выбор обобщающей функции для разрешения конфликтов в значительной степени произволен и определяется тем, с какими данными вы работаете.

## NTILE

Ранги часто используются для формирования групп. Например, пусть требуется найти всех клиентов, общий объем заказов которых занимает место в первых 25%. Следующий запрос использует функцию NTILE для разбиения клиентов на четыре блока (квартиля):

```
SELECT region_id, cust_nbr, SUM(tot_sales) cust_sales,
       NTILE(4) OVER (ORDER BY SUM(tot_sales) DESC) sales_quartile
FROM orders
WHERE year = 2001
GROUP BY region_id, cust_nbr
ORDER BY 4,3 DESC;
```

REGION_ID	CUST_NBR	CUST_SALES	SALES_QUARTILE
9	25	2232703	1
8	17	1944281	1
7	14	1929774	1
5	4	1878275	1
10	26	1808949	1
6	6	1788836	1
8	20	1412006	1
10	27	1322747	1
7	13	1310434	2
7	15	1255591	2
8	18	1253840	2
5	2	1224992	2
9	23	1224992	2
9	24	1224992	2
10	30	1216858	2
6	9	1208959	2
6	10	1196748	3
7	11	1190421	3
7	12	1182275	3
8	19	1174421	3
5	5	1169926	3
5	3	1161286	3
5	1	1151162	3
6	8	1141638	4
8	16	1068467	4
9	22	1036146	4
9	21	1020541	4
10	28	986964	4
6	7	971585	4
10	29	903383	4



Столбец `sales_quartile` запроса содержит функцию `NTILE(4)` для создания четырех блоков. Функция `NTILE` находит положение каждой строки в классификации и относит каждую строку к определенному блоку так, чтобы все блоки содержали одинаковое количество строк. Если количество строк не делится нацело на количество блоков, то дополнительные строки распределяются так, чтобы количество строк блоков отличалось не больше чем на единицу. В предыдущем примере создано четыре блока для 30 строк, при этом первый и второй блоки содержат по восемь строк, а третий и четвертый – по семь. Такие блоки называются *равновысокими* (*equiheight*), так как все они (в идеале) содержат одинаковое количество строк.

Как и в рассмотренном ранее запросе про *N* лучших, если вы хотите отфильтровать ненужные результаты функции `NTILE`, необходимо поместить запрос во встроенное представление:

```
SELECT r.name region, c.name customer, cs.cust_sales
FROM customer c, region r,
  (SELECT region_id, cust_nbr, SUM(tot_sales) cust_sales,
    NTILE(4) OVER (ORDER BY SUM(tot_sales) DESC) sales_quartile
  FROM orders
  WHERE year = 2001
  GROUP BY region_id, cust_nbr) cs
WHERE cs.sales_quartile = 1
  AND cs.cust_nbr = c.cust_nbr
  AND cs.region_id = r.region_id
ORDER BY cs.cust_sales DESC;
```

REGION	CUSTOMER	CUST_SALES
NorthWest US	Worcester Technologies	2232703
SouthWest US	Evans Supply Corp.	1944281
SouthEast US	Madden Industries	1929774
New England	Flowtech Inc.	1878275
Central US	Alpha Technologies	1808949
Mid-Atlantic	Spartan Industries	1788836
SouthWest US	Malden Labs	1412006
Central US	Phillips Labs	1322747

Внешний запрос осуществляет фильтрацию по условию `sales_quartile=1`, удаляя все строки, не относящиеся к лучшей четверти продаж, а затем объединяет клиентов с регионами для получения конечного результата.

## WIDTH\_BUCKET

Подобно функции `NTILE`, `WIDTH_BUCKET` группирует строки результирующего множества в блоки. Но в отличие от `NTILE`, функция `WIDTH_BUCKET` пытается создать блоки *равной ширины* (*equiwidth*), то есть указанный диапазон значений равномерно распределяется по блокам. Если данные имели гауссово распределение, то можно ожидать, что блоки, со-



державшие верхние и нижние ранги, будут состоять из малого количества записей, в то время как блоки, представляющие средние значения, будут содержать много записей.

Появившаяся только в Oracle9i функция `WIDTH_BUCKET` может работать с числовыми типами или датами; она принимает четыре параметра:

1. Выражение, которое формирует блоки.
2. Значение, используемое как начало диапазона для блока номер 1.
3. Значение, используемое как конец диапазона для блока номер N.
4. Количество создаваемых блоков (N).

`WIDTH_BUCKET` использует значения второго, третьего и четвертого параметров для формирования N блоков, содержащих сравнимые ранги. Если выражение выдает значения, не попадающие в диапазон, определяемый вторым и третьим параметром, то функция `WIDTH_BUCKET` формирует два дополнительных блока с номерами 0 и N+1, в которые и помещаются отклонившиеся значения. При желании работать со всем результирующим множеством необходимо убедиться в том, что значения второго и третьего параметров полностью охватывают диапазон значений результирующего множества. Если же вы хотите работать только с подмножеством результата, укажите для второго и третьего параметров соответствующие значения, и все не интересующие вас строки попадут в блоки 0 и N+1.

Для пояснения вышесказанного используем приведенный ранее пример с функцией `NTILE`, чтобы сформировать три блока общих объемов заказов по клиентам:

```
SELECT region_id, cust_nbr,
       SUM(tot_sales) cust_sales,
       WIDTH_BUCKET(SUM(tot_sales), 1, 3000000, 3) sales_buckets
FROM orders
WHERE year = 2001
GROUP BY region_id, cust_nbr
ORDER BY 3;
```

REGION_ID	CUST_NBR	CUST_SALES	SALES_BUCKETS
10	29	903383	1
8	7	971585	1
10	28	986964	1
9	21	1020541	2
9	22	1036146	2
8	16	1068467	2
6	8	1141638	2
5	1	1151162	2
5	3	1161286	2
5	5	1169926	2
8	19	1174421	2
7	12	1182275	2

7	11	1190421	2
6	10	1196748	2
6	9	1208959	2
10	30	1216858	2
5	2	1224992	2
9	24	1224992	2
9	23	1224992	2
8	18	1253840	2
7	15	1255591	2
7	13	1310434	2
10	27	1322747	2
8	20	1412006	2
6	6	1788836	2
10	26	1808949	2
5	4	1878275	2
7	14	1929774	2
8	17	1944281	2
9	25	2232703	3

Для указанных параметров функция `WIDTH_BUCKET` формирует три блока; первый начинается с 1, а третий ограничен максимальным значением, равным 3 000 000. Так как блоков три, значения будут распределены по блокам следующим образом: 1–1 000 000, 1 000 001–2 000 000 и 2 000 001–3 000 000. В итоге в первом блоке оказывается три строки, в третьем – одна, а оставшиеся 26 строк попадают во второй блок.

Значения 1 и 3 000 000 были выбраны для того, чтобы гарантировать попадание всех строк результирующего множества в один из блоков. Если бы требовалось сформировать блоки только для строк, которые имеют обобщенные объемы продаж от \$1 000 000 до \$2 000 000, функция `WIDTH_BUCKET` поместила бы остальные строки в блоки с номерами 0 и 4:

```
SELECT region_id, cust_nbr,
       SUM(tot_sales) cust_sales,
       WIDTH_BUCKET(SUM(tot_sales), 1000000, 2000000, 3) sales_buckets
FROM orders
WHERE year = 2001
GROUP BY region_id, cust_nbr
ORDER BY 3;
```

REGION_ID	CUST_NBR	CUST_SALES	SALES_BUCKETS
-----			
10	29	903383	0
6	7	971585	0
10	28	986964	0
9	21	1020541	1
9	22	1036146	1
8	16	1068467	1
6	8	1141638	1
5	1	1151162	1
5	3	1161286	1
5	5	1169926	1

8	19	1174421	1
7	12	1182275	1
7	11	1190421	1
6	10	1196748	1
6	9	1208959	1
10	30	1216858	1
5	2	1224992	1
9	24	1224992	1
9	23	1224992	1
8	18	1253840	1
7	15	1255591	1
7	13	1310434	1
10	27	1322747	1
8	20	1412006	2
6	6	1788836	3
10	26	1808949	3
5	4	1878275	3
7	14	1929774	3
8	17	1944281	3
9	25	2232703	4

Помните, что функция `WIDTH_BUCKET` не удаляет из результирующего множества строки, не относящиеся к указанному диапазону; они просто помещаются в специальные блоки, которые ваш запрос может как проигнорировать, так и использовать (по вашему желанию).

## CUME\_DIST и PERCENT\_RANK

Две последние ранжирующие функции, `CUME_DIST` и `PERCENT_RANK`, используют ранг указанной строки для вычисления дополнительной информации. Функция `CUME_DIST` (сокращение от Cumulative Distribution — распределение накопленных вероятностей) вычисляет отношение количества строк, имеющих меньший или равный ранг, к общему количеству строк раздела. Функция `PERCENT_RANK` вычисляет отношение ранга строки к количеству строк раздела, используя формулу:

$$(RRP - 1) / (NRP - 1)$$

где:

*RRP* — ранг строки в разделе;

*NRP* — количество строк в разделе.

Оба расчета используют для ранжирования строк функцию `DENSE_RANK`, при этом можно указать как восходящий, так и нисходящий порядок ранжирования. Следующий запрос демонстрирует применение двух этих функций (для обеих задан нисходящий порядок) ко все тому же запросу о годовых объемах заказов клиентов.

```
SELECT region_id, cust_nbr,
       SUM(tot_sales) cust_sales,
```



```

CUME_DIST() OVER (ORDER BY SUM(tot_sales) DESC) sales_cume_dist,
PERCENT_RANK() OVER (ORDER BY SUM(tot_sales) DESC) sales_percent_rank
FROM orders
WHERE year = 2001
GROUP BY region_id, cust_nbr
ORDER BY 3 DESC;

```

REGION_ID	CUST_NBR	CUST_SALES	SALES_CUME_DIST	SALES_PERCENT_RANK
9	25	2232703	.033333333	0
8	17	1944281	.066666667	.034482759
7	14	1929774	.1	.068965517
5	4	1878275	.133333333	.103448276
10	26	1808949	.166666667	.137931034
6	6	1788836	.2	.172413793
8	20	1412006	.233333333	.206896552
10	27	1322747	.266666667	.24137931
7	13	1310434	.3	.275862069
7	15	1255591	.333333333	.310344828
8	18	1253840	.366666667	.344827586
5	2	1224992	.466666667	.379310345
9	23	1224992	.466666667	.379310345
9	24	1224992	.466666667	.379310345
10	30	1216858	.5	.482758621
6	9	1208959	.533333333	.517241379
6	10	1196748	.566666667	.551724138
7	11	1190421	.6	.586206897
7	12	1182275	.633333333	.620689655
8	19	1174421	.666666667	.655172414
5	5	1169926	.7	.689655172
5	3	1161286	.733333333	.724137931
5	1	1151162	.766666667	.75862069
6	8	1141838	.8	.793103448
8	16	1068467	.833333333	.827586207
9	22	1036146	.866666667	.862068966
9	21	1020541	.9	.896551724
10	28	986964	.933333333	.931034483
6	7	971585	.966666667	.965517241
10	29	903383	1	1

Давайте рассмотрим расчеты для клиента номер 1 из предыдущего результирующего множества. По общему объему продаж, равному \$1 151 162, клиенту номер 1 в множестве из 30 клиентов, упорядоченному по убыванию объемов продаж, присвоен ранг 23. Так как всего строк 30, значение CUME\_DIST равно 23/30 или 0,766666667. Функция PERCENT\_RANK выдает  $(23 - 1) / (30 - 1) = 0,75862069$ . Неудивительно, что обе функции возвращают одинаковые значения для строк, имеющих одинаковые значения объемов продаж, ведь вычисления основываются на рангах, которые равны для трех строк.

## Гипотетические функции

В некоторых случаях определить, что *могло* произойти, важнее, чем узнать, что произошло в действительности. В Oracle9i имеются специальные версии функций RANK, DENSE\_RANK, CUME\_DIST и PERCENT\_RANK, которые обеспечивают вычисление рангов и распределений для гипотетических данных, разрешая пользователю увидеть, что произошло бы, если бы указанное значение (или множество значений) было включено в результирующее множество.

Чтобы прояснить ситуацию, давайте сначала ранжируем наших клиентов по объемам продаж за 2001 год, а затем посмотрим, куда в этой классификации попадет гипотетическая личность. Приведем запрос, формирующий ранги и распределения:

```
SELECT cust_nbr, SUM(tot_sales) cust_sales,
       RANK() OVER (ORDER BY SUM(tot_sales) DESC) rank,
       DENSE_RANK() OVER (ORDER BY SUM(tot_sales) DESC) dense_rank,
       CUME_DIST() OVER (ORDER BY SUM(tot_sales) DESC) cume_dist,
       PERCENT_RANK() OVER (ORDER BY SUM(tot_sales) DESC) percent_rank
FROM orders
WHERE year = 2001
GROUP BY cust_nbr
ORDER BY 3;
```

CUST_NBR	CUST_SALES	RANK	DENSE_RANK	CUME_DIST	PERCENT_RANK
25	2232703	1	1	.033333333	.0
17	1944281	2	2	.066666667	.034482759
14	1929774	3	3	.1	.068965517
4	1878275	4	4	.133333333	.103448276
26	1808949	5	5	.166666667	.137931034
6	1788836	6	6	.2	.172413793
20	1412006	7	7	.233333333	.206896552
27	1322747	8	8	.266666667	.24137931
13	1310434	9	9	.3	.275862069
15	1255591	10	10	.333333333	.310344828
18	1253840	11	11	.366666667	.344827586
2	1224992	12	12	.466666667	.379310345
23	1224992	12	12	.466666667	.379310345
24	1224992	12	12	.466666667	.379310345
30	1216858	15	13	.5	.482758621
9	1208959	16	14	.533333333	.517241379
10	1196748	17	15	.566666667	.551724138
11	1190421	18	16	.6	.586206897
12	1182275	19	17	.633333333	.620689655
19	1174421	20	18	.666666667	.655172414
5	1169926	21	19	.7	.689655172
3	1161286	22	20	.733333333	.724137931
1	1151162	23	21	.766666667	.75862069
8	1141638	24	22	.8	.793103448
16	1068467	25	23	.833333333	.827586207



22	1036146	25	24	.866666667	.862068966
21	1020541	27	25	.9	.896551724
28	966964	28	26	.933333333	.931034483
7	971585	29	27	.966666667	.965517241
29	903383	30	28	1	1

Теперь посмотрим, где бы в классификации оказался клиент с объемом продаж, равным одному миллиону долларов:

```
SELECT
  RANK(1000000) WITHIN GROUP
    (ORDER BY SUM(tot_sales) DESC) hyp_rank,
  DENSE_RANK(1000000) WITHIN GROUP
    (ORDER BY SUM(tot_sales) DESC) hyp_dense_rank,
  CUME_DIST(1000000) WITHIN GROUP
    (ORDER BY SUM(tot_sales) DESC) hyp_cume_dist,
  PERCENT_RANK(1000000) WITHIN GROUP
    (ORDER BY SUM(tot_sales) DESC) hyp_percent_rank
FROM orders
WHERE year = 2001
GROUP BY cust_nbr;

HYP_RANK HYP_DENSE_RANK HYP_CUME_DIST HYP_PERCENT_RANK
-----
28          26      .903225806          .9
```

Инструкция WITHIN GROUP как бы вставляет фиктивную строку в результирующее множество перед присвоением рангов. Такую функциональность можно, например, использовать для сравнения реальных объемов продаж с желаемыми.

## Оконные функции

Описанные ранее ранжирующие функции очень полезны при сравнении элементов для фиксированного окна времени, такого как «прошлый год» или «второй квартал». Но что если необходимо выполнить расчеты для окна, которое изменяется по мере продвижения по множеству данных? Оконные функции Oracle позволяют вычислять обобщающие значения для каждой строки результирующего множества для указанного окна. Окно обобщения можно определить одним из трех способов:

- Указав набор строк: «От текущей строки до конца раздела».
- Указав временной интервал: «За 30 дней, предшествующих дате транзакции».
- Указав диапазон значений: «Все строки, количество транзакций которых не превышает 5% от количества транзакций текущей строки».

Для начала сформируем окно, которое занимает весь раздел, а затем посмотрим, как можно отсоединить его от одного или от обоих концов



раздела, чтобы оно скользило вместе с текущей строкой. Все примеры будут базироваться на следующем запросе, который вычисляет общий ежемесячный объем продаж для региона Mid-Atlantic:

```
SELECT month, SUM(tot_sales) monthly_sales
FROM orders
WHERE year = 2001
      AND region_id = 6
GROUP BY month
ORDER BY month;
```

MONTH	MONTHLY_SALES
1	610697
2	428676
3	637031
4	541146
5	592935
6	501485
7	606914
8	460520
9	392898
10	510117
11	532889
12	492458

Сначала просуммируем продажи за месяц для всего результирующего множества, указав «неограниченное» окно. Обратите внимание на использование в примере инструкции ROWS BETWEEN:

```
SELECT month, SUM(tot_sales) monthly_sales,
      SUM(SUM(tot_sales)) OVER (ORDER BY month
      ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) total_sales
FROM orders
WHERE year = 2001
      AND region_id = 6
GROUP BY month
ORDER BY month;
```

MONTH	MONTHLY_SALES	TOTAL_SALES
1	610697	6307766
2	428676	6307766
3	637031	6307766
4	541146	6307766
5	592935	6307766
6	501485	6307766
7	606914	6307766
8	460520	6307766
9	392898	6307766
10	510117	6307766
11	532889	6307766
12	492458	6307766

Каждый раз, когда функция выполняется, она суммирует ежемесячные продажи для месяцев с 1 по 12, то есть одно и то же вычисление производится 12 раз. Это малоэффективный способ генерирования годового объема продаж (более удачный способ будет предложен в разделе «Функции для создания отчетов» далее в этой главе), но он должен дать общее представление о синтаксисе построения окна обобщения. В следующем запросе создадим окно, которое простирается от начала раздела до текущей строки. Функция определяет месяц с максимальным объемом продаж, вплоть до текущего месяца (включительно):

```
SELECT month, SUM(tot_sales) monthly_sales,
       MAX(SUM(tot_sales)) OVER (ORDER BY month
                                ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) max_preceding
FROM orders
WHERE year = 2001
      AND region_id = 6
GROUP BY month
ORDER BY month;
```

MONTH	MONTHLY_SALES	MAX_PRECEDING
1	610697	610697
2	428676	610697
3	637031	637031
4	541146	637031
5	592935	637031
6	501485	637031
7	606914	637031
8	460520	637031
9	392898	637031
10	510117	637031
11	532889	637031
12	492458	637031

В отличие от первого запроса, в котором размер окна фиксирован (12 строк), окно обобщения данного запроса растет от одной строки для первого месяца до 12 строк для 12-го месяца. Ключевое слово `CURRENT ROW` используется для указания того, что окно должно заканчиваться на текущей строке, обследуемой функцией. Если в предыдущем примере заменить `MAX` на `SUM`, будет вычисляться скользящий итог:

```
SELECT month, SUM(tot_sales) monthly_sales,
       SUM(SUM(tot_sales)) OVER (ORDER BY month
                                ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) max_preceding
FROM orders
WHERE year = 2001
      AND region_id = 6
GROUP BY month
ORDER BY month;
```

MONTH	MONTHLY_SALES	MAX_PRECEEDING
1	810697	810697
2	428676	1039373
3	637031	1676404
4	541146	2217550
5	592935	2810485
6	501485	3311970
7	606914	3918884
8	460520	4379404
9	392898	4772302
10	510117	5282419
11	532889	5815308
12	492458	6307766

Мы привели примеры использования окон, которые были зафиксированы с одного или с обоих концов. Теперь же определим окно, которое свободно перемещается с каждой строкой:

```
SELECT month, SUM(tot_sales) monthly_sales,
       AVG(SUM(tot_sales)) OVER (ORDER BY month
                                ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) rolling_avg
FROM orders
WHERE year = 2001
      AND region_id = 6
GROUP BY month
ORDER BY month;
```

MONTH	MONTHLY_SALES	ROLLING_AVG
1	810697	519686.5
2	428676	558801.333
3	637031	535617.667
4	541146	590370.667
5	592935	545188.667
6	501485	567111.333
7	606914	522973
8	460520	486777.333
9	392898	454511.667
10	510117	478634.667
11	532889	511821.333
12	492458	512673.5

Для каждой из 12 строк функция вычисляет средний объем продаж за текущий, предыдущий и последующий месяцы. Таким образом, значение столбца ROLLING\_AVG – это средний объем продаж трехмесячного скользящего окна, симметричного относительно текущего месяца.<sup>1</sup>

<sup>1</sup> Расчеты для первого и двенадцатого месяцев производятся с использованием двухмесячного окна, так как не существует предыдущего месяца для месяца 1 и последующего – для месяца 12.



## FIRST\_VALUE и LAST\_VALUE

Oracle предоставляет две дополнительные обобщающие функции, FIRST\_VALUE и LAST\_VALUE, которые могут использоваться совместно с оконными функциями для определения первого и последнего значений окна. Для запроса, вычисляющего трехмесячное скользящее среднее, можно наряду со средним значением для трех месяцев выводить и значения всех трех месяцев:

```
SELECT month,
       FIRST_VALUE(SUM(tot_sales)) OVER (ORDER BY month
                                         ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) prev_month,
       SUM(tot_sales) monthly_sales,
       LAST_VALUE(SUM(tot_sales)) OVER (ORDER BY month
                                         ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) next_month,
       AVG(SUM(tot_sales)) OVER (ORDER BY month
                                 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) rolling_avg
FROM orders
WHERE year = 2001
      AND region_id = 6
GROUP BY month
ORDER BY month;
```

MONTH	PREV_MONTH	MONTHLY_SALES	NEXT_MONTH	ROLLING_AVG
1	610697	610697	428676	519688.5
2	610697	428676	637031	558801.333
3	428676	637031	541146	535617.667
4	637031	541146	592935	580370.667
5	541146	592935	501485	545188.667
6	592935	501485	606914	567111.333
7	501485	606914	460520	522973
8	606914	460520	392898	486777.333
9	460520	392898	510117	454511.667
10	392898	510117	532889	478634.667
11	510117	532889	492458	511821.333
12	532889	492458	492458	512673.5

Такие функции полезны в запросах, которые сравнивают каждое значение с первым или последним за период, например: «Как соотносится объем продаж каждого месяца с объемом продаж первого месяца?»

## Функции LAG/LEAD

Хотя с технической точки зрения функции LAG и LEAD не являются оконными, они включены в данный раздел, потому что позволяют ссылаться на строки по их позиции относительно текущей строки, во многом подобно инструкциям PRECEDING и FOLLOWING внутри оконных функций. LAG и LEAD удобны при сравнении одной строки результирующего множества с другой строкой того же результирующего множест-

ва. Например, в запросе «Вычислить общий объем продаж по месяцам для региона Mid-Atlantic, включив процент изменения по отношению к предыдущему месяцу» для получения ответа необходимы данные текущей и предыдущей строк. В действительности это двухстрочное окно, но можно задать сдвиг относительно текущей строки как одну или несколько строк, используя LAG и LEAD как специальные оконные функции, в которых работают только внешние концы окна.

Используем функцию LAG для формирования данных, необходимых для ответа на вопрос, заданный в предыдущем абзаце:

```
SELECT month, SUM(tot_sales) monthly_sales,
       LAG(SUM(tot_sales), 1) OVER (ORDER BY month) prev_month_sales
FROM orders
WHERE year = 2001
      AND region_id = 6
GROUP BY month
ORDER BY month;
```

MONTH	MONTHLY_SALES	PREV_MONTH_SALES
1	610697	
2	428676	610697
3	637031	428676
4	541146	637031
5	592935	541146
6	501485	592935
7	606914	501485
8	460520	606914
9	392898	460520
10	510117	392898
11	532889	510117
12	492458	532889

Как и следовало ожидать, значение LAG для месяца 1 содержит NULL, так как предыдущего месяца для него не существует. То же самое справедливо и для значения LEAD месяца 12. Помните об этом, выполняя вычисления, которые используют результат функций LAG и LEAD.

Следующий запрос использует результат предыдущего запроса для получения процентной разницы между объемами продаж от месяца к месяцу. Заметьте, что столбец prev\_month\_sales заключен в функцию NVL, чтобы для месяца 1 при вычислении процентного соотношения не было возвращено NULL:

```
SELECT months.month month, months.monthly_sales monthly_sales,
       ROUND((months.monthly_sales - NVL(months.prev_month_sales,
       months.monthly_sales)) /
       NVL(months.prev_month_sales, months.monthly_sales),
       3) * 100 percent_change
FROM
  (SELECT month, SUM(tot_sales) monthly_sales,
```

```
LAG(SUM(tot_sales), 1) OVER (ORDER BY month) prev_month_sales
FROM orders
WHERE year = 2001
      AND region_id = 6
GROUP BY month) months
ORDER BY month;
```

MONTH	MONTHLY_SALES	PERCENT_CHANGE
1	610697	0
2	428676	-29.8
3	637031	48.6
4	541146	-15.1
5	592935	9.6
6	501485	-15.4
7	606914	21
8	460520	-24.1
9	392898	-14.7
10	510117	29.8
11	532889	4.5
12	492458	-7.6

## Функции для создания отчетов

Подобно описанным ранее оконным функциям, функции для создания отчетов позволяют выполнять различные обобщающие операции (MIN, MAX, SUM, COUNT, AVG и т. д.) над результирующим множеством. Но в отличие от оконных, функции для создания отчетов не могут указывать локальные окна и поэтому выводят результат для всего раздела (или всего результирующего множества, если разделы не заданы). Следовательно, все, что можно сделать посредством функции для создания отчета, можно сделать и с помощью оконной функции, только первый вариант обычно будет более эффективным.

Ранее в этой главе оконная функция с неограниченным окном использовалась для вывода общих объемов продаж для 12 месяцев 2001 года:

```
SELECT month,
       SUM(tot_sales) monthly_sales,
       SUM(SUM(tot_sales)) OVER (ORDER BY month
                                ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) total_sales
FROM orders
WHERE year = 2001
      AND region_id = 6
GROUP BY month
ORDER BY month;
```

MONTH	MONTHLY_SALES	TOTAL_SALES
1	610697	6307766
2	428676	6307766



3	637031	6307766
4	541146	6307766
5	592935	6307766
6	501485	6307766
7	606914	6307766
8	460520	6307766
9	392898	6307766
10	510117	6307766
11	532889	6307766
12	492458	6307766

Следующий запрос для порождения тех же самых результатов использует функцию создания отчета:

```
SELECT month,
       SUM(tot_sales) monthly_sales,
       SUM(SUM(tot_sales)) OVER (ORDER BY month
                                ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) window_sales,
       SUM(SUM(tot_sales)) OVER () reporting_sales
FROM orders
WHERE year = 2001
      AND region_id = 6
GROUP BY month
ORDER BY month;
```

MONTH	MONTHLY_SALES	WINDOW_SALES	REPORTING_SALES
1	610697	6307766	6307766
2	428676	6307766	6307766
3	637031	6307766	6307766
4	541146	6307766	6307766
5	592935	6307766	6307766
6	501485	6307766	6307766
7	606914	6307766	6307766
8	460520	6307766	6307766
9	392898	6307766	6307766
10	510117	6307766	6307766
11	532889	6307766	6307766
12	492458	6307766	6307766

Пустые скобки после инструкции OVER в столбце reporting\_sales показывают, что в суммировании должно участвовать все результирующее множество; такой же эффект достигается применением неограниченной оконной функции. Будем надеяться, что вы согласны с тем, что функция создания отчета проще для понимания, чем неограниченная оконная функция.

Функции создания отчета полезны в тех случаях, когда для ответа на бизнес-запрос вам необходимы как подробные, так и суммарные данные (или данные разных уровней обобщения). Например, ответ на запрос «Вывести ежемесячные объемы продаж 2001 года с указанием

для каждого месяца соответствующего процента годовых продаж» требует детальных данных для обобщения сначала на уровне месяца, а затем на уровне года. Вместо того чтобы выполнять два суммирования на основе частных данных, используем функцию SUM с инструкцией GROUP BY для обобщения на уровне месяца, а затем применим функцию создания отчета для суммирования ежемесячных итогов:

```
SELECT month,
       SUM(tot_sales) monthly_sales,
       SUM(SUM(tot_sales)) OVER () yearly_sales
FROM orders
WHERE year = 2001
GROUP BY month
ORDER BY month;
```

MONTH	MONTHLY_SALES	YEARLY_SALES
1	3028325	39593192
2	3289336	39593192
3	3411024	39593192
4	3436482	39593192
5	3749264	39593192
6	3204730	39593192
7	3233532	39593192
8	3081290	39593192
9	3388292	39593192
10	3279637	39593192
11	3187858	39593192
12	3323422	39593192

Затем просто разделим MONTHLY\_SALES на YEARLY\_SALES, чтобы вычислить запрошенные проценты (см. раздел «RATIO\_TO\_REPORT» далее в этой главе).

## Разбиение отчетов

Как и ранжирующие функции, функции для создания отчета могут содержать инструкцию PARTITION BY для разбиения результирующего множества на несколько частей, что позволяет выполнить несколько суммирований для различных подмножеств результирующего множества. Следующий запрос выводит общий объем продаж для каждого продавца по регионам, а также для сравнения общий объем продаж региона:

```
SELECT region_id, salesperson_id,
       SUM(tot_sales) sp_sales,
       SUM(SUM(tot_sales)) OVER (PARTITION BY region_id) region_sales
FROM orders
WHERE year = 2001
GROUP BY region_id, salesperson_id
```

ORDER BY region\_id, salesperson\_id;

REGION_ID	SALESPERSON_ID	SP_SALES	REGION_SALES
5	1	1927580	6585641
5	2	1461898	6585641
5	3	1501039	6585641
5	4	1695124	6585641
6	5	1688252	6307766
6	6	1392648	6307766
6	7	1458053	6307766
6	8	1768813	6307766
7	9	1735575	6868495
7	10	1723305	6868495
7	11	1737093	6868495
7	12	1672522	6868495
8	13	1516776	6853015
8	14	1814327	6853015
8	15	1760098	6853015
8	16	1761814	6853015
9	17	1710831	6739374
9	18	1625456	6739374
9	19	1645204	6739374
9	20	1757883	6739374
10	21	1542152	6238901
10	22	1468316	6238901
10	23	1443837	6238901
10	24	1784596	6238901

Значение столбца `REGION_SALES` одинаково для всех торговых представителей одного региона. В следующем разделе вы увидите два способа использования этой информации для вычисления процентных соотношений.

## RATIO\_TO\_REPORT

Функции создания отчетов также часто используются для получения значения общего знаменателя при вычислении производительности. Например, следующим логическим шагом для запроса из предыдущего раздела было бы деление общего объема продаж каждого продавца (`SP_SALES`) на общий объем продаж региона (`REGION_SALES`) для определения того, какой процент общих продаж региона принадлежит данному продавцу. В качестве знаменателя при вычислении процента можно использовать функцию создания отчета:

```
SELECT region_id, salesperson_id,
       SUM(tot_sales) sp_sales,
       ROUND(SUM(tot_sales) /
             SUM(SUM(tot_sales)) OVER (PARTITION BY region_id),
             2) percent_of_region
FROM orders
```



```
WHERE year = 2001
GROUP BY region_id, salesperson_id
ORDER BY region_id, salesperson_id 1,2;
```

REGION_ID	SALESPERSON_ID	SP_SALES	PERCENT_OF_REGION
5	1	1927580	.29
5	2	1461898	.22
5	3	1501039	.23
5	4	1695124	.26
6	5	1688252	.27
6	6	1392648	.22
6	7	1458053	.23
6	8	1768813	.28
7	9	1735575	.25
7	10	1723305	.25
7	11	1737093	.25
7	12	1672522	.24
8	13	1516776	.22
8	14	1814327	.26
8	15	1760098	.26
8	16	1761814	.26
9	17	1710831	.25
9	18	1625456	.24
9	19	1645204	.24
9	20	1757883	.26
10	21	1542152	.25
10	22	1468316	.24
10	23	1443837	.23
10	24	1784596	.29

Но поскольку такая операция используется очень часто, Oracle избавляет нас от проблем с помощью функции `RATIO_TO_REPORT`. Функция `RATIO_TO_REPORT` позволяет вычислить вклад каждой строки или во все результирующее множество, или в некоторое его подмножество (если использована инструкция `PARTITION BY`). Следующий запрос использует функцию `RATIO_TO_REPORT` для получения процентного вклада каждого продавца в общий объем продаж своего региона:

```
SELECT region_id, salesperson_id,
       SUM(tot_sales) sp_sales,
       ROUND(RATIO_TO_REPORT(SUM(tot_sales))
             OVER (PARTITION BY region_id, 2) sp_ratio
FROM orders
WHERE year = 2001
GROUP BY region_id, salesperson_id
ORDER BY 1,2;
```

REGION_ID	SALESPERSON_ID	SP_SALES	SP_RATIO
5	1	1927580	.29
5	2	1461898	.22

5	3	1501039	.23
5	4	1695124	.26
6	5	1688252	.27
6	6	1392648	.22
6	7	1458053	.23
6	8	1768813	.28
7	9	1735575	.25
7	10	1723305	.25
7	11	1737093	.25
7	12	1672522	.24
8	13	1516776	.22
8	14	1814327	.26
8	15	1760098	.26
8	16	1761814	.26
9	17	1710831	.25
9	18	1625456	.24
9	19	1645204	.24
9	20	1757883	.26
10	21	1542152	.25
10	22	1468316	.24
10	23	1443837	.23
10	24	1784596	.29

## Резюме

В этой главе представлено очень много материала, поэтому не расстраивайтесь, если вам потребуется перечитать ее несколько раз, чтобы почувствовать все эти разнообразные аналитические функции и осознать, как их следует применять. Функций очень много, поэтому будет проще, если вы сосредоточитесь сначала на какой-то одной категории (ранжирующие функции, оконные или функции создания отчетов), а потом перейдете к следующей. Те из вас, кто уже много лет работают с Oracle, вероятно, не сразу решатся опробовать эти функции. Аналитические функции Oracle компактны и эффективны, кроме того, они выполняют аналитические расчеты там, откуда они родом – на сервере базы данных, вместо того чтобы прибегать к процедурным языкам или табличным макросам.

# 14

## Советы умудренных опытом

Для написания эффективного SQL-кода необходим опыт. Любой запрос можно написать несколькими способами, при этом один из них может оказаться в сотни раз медленнее другого. В этой главе будет дано несколько советов и предложено несколько идей, которые помогут сделать ваши операторы SQL более эффективными.

### Когда и какие конструкции использовать?

В зависимости от обстоятельств использование одной конструкции SQL является более предпочтительным, чем другой. Например, иногда лучше использовать оператор EXISTS вместо IN. Но для NOT EXISTS и NOT IN это уже не так. В следующем разделе будут описаны эти конструкции.

### EXISTS вместо DISTINCT

Использование ключевого слова DISTINCT в инструкции SELECT приводит к удалению из результирующего множества повторяющихся строк. Для удаления дубликатов Oracle выполняет сортировку, которая требует времени и определенного дискового пространства. Так что если присутствие в результирующем множестве повторяющихся строк допустимо, старайтесь не использовать DISTINCT. Если же дубликаты недопустимы или приложение не может их обработать, используйте вместо DISTINCT оператор EXISTS.

Предположим, что нужно вывести фамилии клиентов, сделавших заказы. Запрос базируется на двух таблицах: CUSTOMER и CUST\_ORDER. Запрос, использующий DISTINCT, будет выглядеть следующим образом:

```
SELECT DISTINCT C.CUST_NBR, C.NAME
```



```
FROM CUSTOMER C, CUST_ORDER O
WHERE C.CUST_NBR = O.CUST_NBR;
```

Посмотрим на план выполнения этого запроса. Обратите внимание на операцию SORT, которая является результатом применения DISTINCT.

```
Query Plan
-----
SELECT STATEMENT   Cost = 3056
  SORT UNIQUE
    MERGE JOIN
      INDEX FULL SCAN IND_ORD_CUST_NBR
    SORT JOIN
      TABLE ACCESS FULL CUSTOMER
```

Перепишем запрос так, чтобы в нем использовался оператор EXISTS:

```
SELECT C.CUST_NBR, C.NAME
FROM CUSTOMER C
WHERE EXISTS (SELECT 1 FROM CUST_ORDER O WHERE C.CUST_NBR = O.CUST_NBR);
```

Обратимся к плану выполнения новой версии запроса. Заметьте, как изменилась стоимость запроса (cost) по сравнению с предыдущей версией, использовавшей DISTINCT:

```
Query Plan
-----
SELECT STATEMENT   Cost = 320
  FILTER
    TABLE ACCESS FULL CUSTOMER
    INDEX RANGE SCAN IND_ORD_CUST_NBR
```

Затраты на выполнение запроса с EXISTS составляют менее одной девятой затрат на выполнение версии с DISTINCT. Все дело в том, что удалось избежать сортировки.

## EXISTS и IN

Во многих книгах по SQL обсуждается факт лучшей производительности конструкции NOT EXISTS по сравнению с NOT IN. Наш опыт показывает, что для Oracle8i EXPLAIN PLAN, формируемый для NOT EXISTS, полностью эквивалентен формируемому для NOT IN, и производительность двух операторов совпадает.

Но вот сравнение EXISTS и IN – это отдельная история. В некоторых случаях EXISTS работает лучше, чем IN. Давайте рассмотрим такой пример. Используем IN для удаления заказов клиентов региона 5:

```
DELETE FROM CUST_ORDER
WHERE CUST_NBR IN
(SELECT CUST_NBR FROM CUSTOMER
WHERE REGION_ID = 5);
```

**План выполнения запроса таков:**

```
Query Plan
-----
DELETE STATEMENT   Cost = 3
  DELETE CUST_ORDER
    HASH JOIN
      TABLE ACCESS FULL CUST_ORDER
      TABLE ACCESS FULL CUSTOMER
```

**Теперь посмотрим на этот же запрос, но использующий EXISTS:**

```
DELETE FROM CUST_ORDER
WHERE EXISTS
(SELECT CUST_NBR FROM CUSTOMER
WHERE CUST_ORDER.CUST_NBR = CUSTOMER.CUST_NBR
AND REGION_ID = 5);
```

**Обратимся к плану выполнения EXISTS-версии:**

```
Query Plan
-----
DELETE STATEMENT   Cost = 1
  DELETE CUST_ORDER
    FILTER
      TABLE ACCESS FULL CUST_ORDER
      TABLE ACCESS BY INDEX ROWID CUSTOMER
      INDEX UNIQUE SCAN CUSTOMER_PK
```

Отметьте разницу стоимостей запросов. Версия IN имеет стоимость 3, а EXISTS-версия – всего 1. Если используется инструкция EXISTS, то план выполнения управляется внешней таблицей, в то время как при использовании инструкции IN планом выполнения управляет таблица подзапроса. Запрос с EXISTS будет почти всегда быстрее, чем IN-запрос, за исключением тех случаев, когда таблица подзапроса имеет гораздо меньше строк, чем внешняя таблица.

## WHERE и HAVING

В главе 4 рассказывалось об инструкциях GROUP BY и HAVING. Иногда при написании запроса GROUP BY необходимо задать условие, которое может быть записано и в инструкции WHERE, и в HAVING. Если у вас есть выбор, указывайте условие в инструкции WHERE, так вы улучшите производительность запроса. Удаление строк до выполнения вычислений – это менее дорогостоящая операция, чем удаление результатов.

Давайте на примере рассмотрим преимущества WHERE над HAVING. Напишем запрос с инструкцией HAVING, который выводит количество заказов в 2000 году:

```
SELECT YEAR, COUNT(*)
FROM ORDERS
```

```
GROUP BY YEAR
HAVING YEAR = 2000;

YEAR    COUNT(*)
-----
2000    720
```

План выполнения данного запроса таков:

```
Query Plan
-----
SELECT STATEMENT   Cost = 6
  FILTER
    SORT GROUP BY
      INDEX FAST FULL SCAN ORDERS_PK
```

Теперь посмотрим на тот же самый запрос, в котором ограничение на год задано в инструкции WHERE:

```
SELECT YEAR, COUNT(*)
FROM ORDERS
WHERE YEAR = 2000
GROUP BY YEAR;

YEAR    COUNT(*)
-----
2000    720
```

Обратимся к его плану выполнения:

```
Query Plan
-----
SELECT STATEMENT   Cost = 2
  SORT GROUP BY NOSORT
    INDEX FAST FULL SCAN ORDERS_PK
```

Запрос с инструкцией HAVING сначала выполняет групповую операцию, а затем фильтрует группы по указанному условию. WHERE-версия запроса фильтрует строки до выполнения групповой операции. Результатом фильтрации является уменьшение количества строк обобщения и, следовательно, увеличение производительности запроса.

Но знайте, что инструкция WHERE может обеспечить не все типы фильтрации. Иногда может потребоваться сначала просуммировать данные, а потом выполнять фильтрацию итоговых данных на основе суммарных значений. В таких ситуациях приходится использовать для фильтрации инструкцию HAVING, так как только HAVING «видит» обобщенные данные. Кроме того, возможны и такие случаи, когда для обеспечения соответствующей фильтрации данных необходимо использовать в запросе и инструкцию WHERE, и инструкцию HAVING. Подробную информацию об этом можно найти в главе 4.



## UNION и UNION ALL

Операции UNION и UNION ALL обсуждались в главе 7. UNION ALL объединяет результаты двух операторов SELECT. UNION объединяет результаты двух операторов SELECT и возвращает только неповторяющиеся строки объединения, удаляя дубликаты. Очевидно, что для удаления дубликатов UNION выполняет дополнительную операцию (по сравнению с UNION ALL). Этой дополнительной операцией является сортировка, которая снижает производительность. Поэтому если приложение может обрабатывать дубликаты или вы уверены, что повторений в результирующем множестве не будет, используйте UNION ALL вместо UNION.

Чтобы осознать вышесказанное, давайте рассмотрим пример. Следующий запрос использует UNION для вывода списка заказов, которые отпускаются по цене, превышающей \$50, либо которые сделаны клиентом, проживающим в регионе 5.

```
SELECT ORDER_NBR, CUST_NBR FROM CUST_ORDER WHERE SALE_PRICE > 50
UNION
SELECT ORDER_NBR, CUST_NBR FROM CUST_ORDER
WHERE CUST_NBR IN
(SELECT CUST_NBR FROM CUSTOMER WHERE REGION_ID = 5);
```

ORDER_NBR	CUST_NBR
1000	1
1001	1
1002	5
1003	4
1004	4
1005	8
1006	1
1007	5
1008	5
1009	1
1011	1
1012	1
1015	5
1017	4
1019	4
1021	8
1023	1
1025	5
1027	5
1029	1

20 rows selected.

План выполнения запроса UNION таков:

Query Plan

---

SELECT STATEMENT Cost = 8

```

SORT UNIQUE
UNION-ALL
TABLE ACCESS FULL CUST_ORDER
HASH JOIN
TABLE ACCESS FULL CUSTOMER
TABLE ACCESS FULL CUST_ORDER

```

Теперь применим для получения той же самой информации UNION ALL, а не UNION:

```

SELECT ORDER_NBR, CUST_NBR FROM CUST_ORDER WHERE SALE_PRICE > 50
UNION ALL
SELECT ORDER_NBR, CUST_NBR FROM CUST_ORDER
WHERE CUST_NBR IN
(SELECT CUST_NBR FROM CUSTOMER WHERE REGION_ID = 5);

```

ORDER_NBR	CUST_NBR
1001	1
1003	4
1005	8
1009	1
1012	1
1017	4
1021	8
1029	1
1001	1
1000	1
1002	5
1003	4
1004	4
1006	1
1007	5
1008	5
1009	1
1012	1
1011	1
1015	5
1017	4
1019	4
1023	1
1025	5
1027	5
1029	1

26 rows selected.

Как видите, в выходных данных присутствуют повторения. Однако UNION ALL работает эффективнее, чем UNION, что видно из его плана выполнения:

Query Plan

```

-----
SELECT STATEMENT      Cost = 4

```

```
UNION-ALL  
TABLE ACCESS FULL CUST_ORDER  
HASH JOIN  
TABLE ACCESS FULL CUSTOMER  
TABLE ACCESS FULL CUST_ORDER
```

В данном плане стоимость запроса равна 4, а в предыдущем – 8. Дополнительная операция (SORT UNIQUE) запроса UNION делает его более медленным, чем UNION ALL.

## Избегайте ненужного разбора операторов

Прежде чем ваш SQL-код будет выполнен Oracle, он должен быть проанализирован. Важность разбора в терминах настройки SQL связана с тем, что неважно, сколько раз данный оператор SQL будет выполнен, проанализирован он должен быть только единожды. В процессе разбора выполняются следующие шаги (необязательно в приведенной последовательности):

- Проверяется синтаксис оператора SQL.
- Словарь данных просматривается для проверки определений таблиц и столбцов.
- Словарь данных просматривается для проверки прав доступа к соответствующим объектам.
- Устанавливаются блокировки разбора на требуемые объекты.
- Определяется оптимальный план выполнения.
- Оператор загружается в разделяемую область SQL (также называемую библиотечным кэшем) разделяемого пула системной глобальной области (SGA, system global area). План выполнения и информация о разборе оператора сохраняются и затем используются, если этот же оператор будет выполнен вновь.

Если оператор SQL затрагивает удаленные объекты (например, используется соединение с внешней базой данных), эти шаги повторяются для удаленных объектов. Как видите, в ходе анализа выполняется огромная работа. Однако разбор оператора выполняется, только если Oracle не находит идентичного оператора SQL в совместно используемой области (библиотечном кэше) SGA.

Перед разбором оператора SQL Oracle просматривает библиотечный кэш в поисках идентичного оператора. Если обнаружено точное совпадение, нет необходимости в повторном разборе оператора. Но если идентичный оператор SQL не найден, Oracle проделывает все шаги, перечисленные выше.

Ключевым в последнем абзаце является слово «идентичный». Чтобы разделять одну область SQL, два оператора должны быть идентичными в полном смысле слова. Два оператора, которые похожи друг на



друга или возвращают одинаковые результаты, не обязательно идентичны. Чтобы считаться идентичными, два оператора должны удовлетворять следующим условиям:

- Иметь одинаковые символы с учетом регистра.
- Иметь одинаковые символы-разделители и символы перехода на новую строку.
- Ссылаться на одинаковые объекты, используя одинаковые названия, которые, в свою очередь, должны иметь одинаковых владельцев.

Если есть вероятность того, что ваше приложение будет несколько раз выполнять одинаковые (или похожие) операторы SQL, старайтесь всеми силами избегать ненужных разборов. Это повысит общую производительность приложения. Для уменьшения количества разборов можно применять следующие приемы:

- Использование связанных переменных.
- Использование псевдонимов таблиц.

## Использование связанных переменных

Когда приложение используют несколько пользователей, они фактически снова и снова выполняют один и тот же набор операторов SQL, только с разными значениями данных. Например, один сотрудник отдела обслуживания может выполнить такой запрос:

```
SELECT * FROM CUSTOMER WHERE CUST_NBR = 121;
```

а другой сотрудник того же отдела выполнит следующий запрос:

```
SELECT * FROM CUSTOMER WHERE CUST_NBR = 328;
```

Эти два оператора похожи, но не идентичны – значения идентификаторов клиентов не совпадают, поэтому Oracle должен выполнять разбор дважды.

Так как единственным отличием двух операторов является значение номера клиента, можно переписать приложение, используя связанные переменные. Тогда рассматриваемый оператор SQL мог бы выглядеть следующим образом:

```
SELECT * FROM CUSTOMER WHERE CUST_NBR = :X;
```

Такой оператор Oracle разбирался бы только один раз. Реальные номера клиентов подставлялись бы после разбора при выполнении оператора. Несколько параллельно работающих программ могли бы совместно использовать один экземпляр данного оператора SQL, подставляя различные номера клиентов.

В многопользовательских приложениях ситуации, подобные вышеописанной, встречаются очень часто, и использование связанных пе-

ременных может значительно увеличить общую производительность (за счет отказа от ненужных разборов).

## Использование псевдонимов таблиц

Использование псевдонимов таблиц может повысить эффективность выполнения операторов SQL. Прежде чем вдаваться в тонкости влияния псевдонимов таблиц на производительность, давайте вспомним, что представляют собой эти псевдонимы и как они используются.

При выборке данных из двух или более таблиц необходимо указывать, какой из таблиц принадлежит каждый столбец. Иначе, если две таблицы содержат столбцы с одинаковыми названиями, будет выдано сообщение об ошибке:

```
SELECT CUST_NBR, NAME, ORDER_NBR
FROM CUSTOMER, CUST_ORDER;
SELECT CUST_NBR, NAME, ORDER_NBR
      *
ERROR at line 1:
ORA-00918: column ambiguously defined
```

В данном случае ошибка вызвана тем, что и таблица CUSTOMER, и таблица CUST\_ORDER содержат столбец CUST\_NBR. Oracle не знает, на какой из них вы ссылаетесь. Чтобы исправить положение, можно переписать оператор следующим образом:

```
SELECT CUSTOMER.CUST_NBR, CUSTOMER.NAME, CUST_ORDER.ORDER_NBR
FROM CUSTOMER, CUST_ORDER
WHERE CUSTOMER.CUST_NBR = CUST_ORDER.CUST_NBR;
```

CUST_NBR	NAME	ORDER_NBR
1	Cooper Industries	1001
1	Cooper Industries	1000
5	Gentech Industries	1002
4	Flowtech Inc.	1003
4	Flowtech Inc.	1004
8	Zantech Inc.	1005
1	Cooper Industries	1006
5	Gentech Industries	1007
5	Gentech Industries	1008
1	Cooper Industries	1009
1	Cooper Industries	1012
1	Cooper Industries	1011
5	Gentech Industries	1015
4	Flowtech Inc.	1017
4	Flowtech Inc.	1019
8	Zantech Inc.	1021
1	Cooper Industries	1023
5	Gentech Industries	1025

5 Gentech Industries	1027
1 Cooper Industries	1029

20 rows selected.

Обратите внимание на использование названия таблицы для уточнения названия каждого столбца. Это устраняет двусмысленность, возникшую при ссылке на столбец CUST\_NBR.

Вместо того чтобы указывать для столбцов полные названия таблиц, можно использовать псевдонимы, например:

```
SELECT C.CUST_NBR, C.NAME, O.ORDER_NBR
FROM CUSTOMER C, CUST_ORDER O
WHERE C.CUST_NBR = O.CUST_NBR;
```

CUST_NBR	NAME	ORDER_NBR
1	Cooper Industries	1001
1	Cooper Industries	1000
5	Gentech Industries	1002
4	Flowtech Inc.	1003
4	Flowtech Inc.	1004
8	Zantech Inc.	1005
1	Cooper Industries	1006
5	Gentech Industries	1007
5	Gentech Industries	1008
1	Cooper Industries	1009
1	Cooper Industries	1012
1	Cooper Industries	1011
5	Gentech Industries	1015
4	Flowtech Inc.	1017
4	Flowtech Inc.	1019
8	Zantech Inc.	1021
1	Cooper Industries	1023
5	Gentech Industries	1025
5	Gentech Industries	1027
1	Cooper Industries	1029

20 rows selected.

Буквы «С» и «О» – это псевдонимы таблиц. Вы можете указать псевдонимы таблиц за соответствующими названиями таблиц в инструкции FROM, и затем они могут использоваться в запросе вместо названий таблиц. Псевдонимы таблиц представляют собой удобную краткую запись, что позволяет запросам быть более читабельными и лаконичными.



Длина псевдонимов таблиц не ограничена одним символом, она может достигать 30 символов.

Еще один факт, о котором необходимо помнить при использовании псевдонимов таблиц, заключается в том, что если в инструкции FROM



определены псевдонимы таблиц, далее следует использовать только эти псевдонимы, а не реальные названия таблиц. Если вы указываете для таблицы псевдоним, а затем используете в запросе фактическое название таблицы, то будет выдано сообщение об ошибке:

```
SELECT C.CUST_NBR, C.NAME, O.ORDER_NBR
FROM CUSTOMER C, CUST_ORDER O
WHERE CUSTOMER.CUST_NBR = CUST_ORDER.CUST_NBR;
WHERE CUSTOMER.CUST_NBR = CUST_ORDER.CUST_NBR
```

```
ERROR at line 3:
ORA-00904: invalid column name
```

Столбец CUST\_NBR присутствует в таблицах CUSTOMER и CUST\_ORDER. Без необходимого уточнения столбец будет считаться неоднозначно определенным. Следовательно, необходимо указывать столбец CUST\_NBR с псевдонимом таблицы (или с полным названием таблицы, если вы не применяете псевдонимы). Однако остальные два столбца, используемые запросом, определяются однозначно. Поэтому оператор, в котором уточнение приведено только для столбца CUST\_NBR, вполне работоспособен:

```
SELECT C.CUST_NBR, NAME, ORDER_NBR
FROM CUSTOMER C, CUST_ORDER O
WHERE C.CUST_NBR = O.CUST_NBR;
```

CUST_NBR	NAME	ORDER_NBR
1	Cooper Industries	1001
1	Cooper Industries	1000
5	Gentech Industries	1002
4	Flowtech Inc.	1003
4	Flowtech Inc.	1004
8	Zantech Inc.	1005
1	Cooper Industries	1006
5	Gentech Industries	1007
5	Gentech Industries	1008
1	Cooper Industries	1009
1	Cooper Industries	1012
1	Cooper Industries	1011
5	Gentech Industries	1015
4	Flowtech Inc.	1017
4	Flowtech Inc.	1019
8	Zantech Inc.	1021
1	Cooper Industries	1023
5	Gentech Industries	1025
5	Gentech Industries	1027
1	Cooper Industries	1029

20 rows selected.

Пришло время поговорить о псевдонимах таблиц с точки зрения производительности. Так как запрос не уточняет столбцы NAME и ORDER\_NBR, Oracle, разбирая оператор, должен просматривать обе таблицы CUSTOMER и CUST\_ORDER для того, чтобы определить, какой из таблиц принадлежит каждый из столбцов. Для одного запроса можно пренебречь временем, потраченным на такой поиск, но если предстоит разобрать множество подобных запросов, накопится значительное время. Хорошей программистской привычкой является указание псевдонимов таблиц для *всех* столбцов запроса, даже если они однозначно определены, чтобы избавить Oracle от дополнительного поиска при разборе оператора.

## Применяйте полностью определенный SQL для систем поддержки принятия решений

Мы только что говорили о преимуществах использования связанных переменных. Применение связанных переменных часто благотворно влияет на производительность. Но необходимо упомянуть и об одном недостатке. Связанные переменные скрывают от оптимизатора реальные значения. Такое утаивание может иметь и негативные последствия, особенно для производительности систем поддержки принятия решений. Рассмотрим, например, такой оператор:

```
SELECT * FROM CUSTOMER WHERE REGION_ID = :X
```

Оптимизатор может разобрать данный оператор, но не может принять в расчет реальный выбираемый регион. Если 90% клиентов проживает в регионе с номером 5, то при выборе таких клиентов наиболее эффективным подходом был бы полный просмотр таблицы. Для выбора клиентов других регионов больше подошел бы просмотр индекса. Когда вы жестко кодируете значения в операторах SQL, оптимизатор, оценивающий стоимость, может посмотреть на гистограммы (разновидность статистики) и сформировать план выполнения, который принимает в расчет конкретные значения. При использовании связанных переменных оптимизатор формирует план выполнения, не имея полной картины оператора SQL. Такой план может быть самым эффективным, а может и нет.

В системах поддержки принятия решений (DSS) очень редко случается так, что несколько пользователей снова и снова используют один и тот же запрос. Обычно небольшое количество пользователей выполняет различные сложные запросы к большой базе данных. Так как операторы SQL будут повторяться очень редко, время, сохраненное за счет использования связанных переменных, будет незначительным. В то же время, так как приложения DSS выполняют сложные запросы к большим базам данных, значительным может быть время, необходимое для выборки результирующих данных. Поэтому важно, чтобы оптимизатор формировал для запроса наиболее эффективный план вы-



полнения. Чтобы помочь оптимизатору создать такой план, необходимо снабдить его как можно большим количеством информации, в том числе реальными значениями столбцов или переменных. Поэтому в приложениях DSS следует использовать полностью определенные операторы SQL с жестко закодированными значениями вместо связанных переменных.

Данная ранее рекомендация по применению связанных переменных остается в силе для приложений OLTP (Online Transaction Processing – оперативная обработка транзакций). В системах OLTP несколько пользователей одновременно применяют одни и те же программы, выполняя одни и те же запросы. Объем данных, возвращаемых запросом, обычно достаточно мал. Поэтому время разбора является более важным фактором для производительности, чем в системах DSS. При разработке приложений OLTP используйте связанные переменные для того, чтобы сберечь время разбора и пространство разделяемой области SQL.



# Алфавитный указатель

## Специальные символы

- > (больше чем), оператор, 41
  - оператор самообъединения не по равенству, 63
- >= (больше или равно), оператор, 41
- (вычитание), оператор даты, 140
- < (меньше чем), оператор, 41
  - оператор самообъединения не по равенству, 63
- <= (меньше или равно), оператор, 41
- != (неравенство), оператор, 39
- (+), оператор внешнего объединения, 51
  - внешние самообъединения, 60
- \_ (подчеркивание), специальный символ, 41
- = (равенство), оператор, 39
- () (скобки)
  - подзапросы, 91
  - приоритет условий/операторов, 45
- %, специальный символ, 41

## А

- A.D., индикатор (формат года), 130
- ADD\_MONTHS, функция, 139
  - вычитание дат, 141
- ALL, ключевое слово
  - многострочные подзапросы, 94
  - обобщающие функции, 79, 81
- ALL\_UPDATABLE\_COLUMNS,
  - представление словаря данных, 70
- ALTER DATABASE, команда,
  - часовые пояса, 153
- ALTER SESSION, команда,
  - форматирование дат, 134
- A.M., индикатор (формат времени), 131
- AND, логический оператор
  - в инструкции WHERE, 36

ANSI (American National Standards Institute), 21

- литералы дат, 135
- обход деревьев, 194
- синтаксис объединения, 71
  - внешние объединения, 76
  - преимущества, 76
- ANY, ключевое слово,
  - многострочные подзапросы, 94
- apply\_split, процедура, 238
  - объектные таблицы, 240

## В

- V.C., индикатор (формат года), 130
- BETWEEN, оператор, 40

## С

- CASE, выражение
  - if-then-else, функциональность, 211
- UPDATE, оператор, 217
  - выборочное выполнение функции, 216
  - выборочное обобщение, 220, 222
  - необязательные обновления, 218
  - ошибки деления на ноль, 222
  - преимущества, 211
  - преобразование результирующих множеств, 215
  - простое, 213
  - с поиском, 212
  - управление состоянием, 224
- CAST, функция, 169
- CHAR, тип данных,
  - функция TO\_DATE, 123
- CONNECT BY, инструкция,
  - иерархические запросы, 109
- CREATE TABLE, оператор,
  - объектные таблицы, 239

CREATE TYPE BODY, оператор, 238  
CUBE, ключевое слово  
    частичная операция, 281  
CUME\_DIST, аналитическая функция,  
    324  
CURRENT ROW, ключевое слово  
    оконные функции, 329  
CURRENT\_DATE, функция, 165  
CURRENT\_TIMESTAMP, функция,  
    165

## D

DATE, тип данных  
    внутренний формат хранения, 122  
    преобразование, 122  
        ошибки, 125  
    указание формата, 124  
    формат по умолчанию, 123  
DBA\_UPDATABLE\_COLUMNS,  
    представление словаря данных, 70  
DBTIMEZONE, ключевое слово, 154  
DBTIMEZONE, функция, 164  
DDL (Data Definition Language), 19  
DECODE, функция  
    UPDATE, оператор, 217  
    выборочное выполнение функции,  
        216  
    выборочное обобщение, 220, 222  
    необязательные обновления, 218  
    ошибки, 209  
    ошибки деления на ноль, 222  
    преобразование результирующих  
        множеств, 214  
    синтаксис, 207, 208  
    управление состоянием, 223  
DELETE, оператор, 31  
    в DML, 20  
    многостолбцовые подзапросы, 98  
    представления объединений, 68  
    скалярные подзапросы, 94  
DENSE\_RANK, аналитическая  
    функция, 320  
DISTINCT, ключевое слово, 29  
    обобщающие функции, 81  
    самообъединения не по равенству,  
        62  
    сравнение с EXISTS, 339  
DML (Data Manipulation Language), 19  
    DELETE, оператор, 31  
    INSERT, оператор, 30

SELECT, оператор, 26  
    DISTINCT, ключевое слово, 29  
    ORDER BY, инструкция, 28  
    WHERE, инструкция, 25  
    элементы инструкции, 28  
UPDATE, оператор, 31  
операторы  
    встроенные представления, 112  
    представления объединений и,  
        71  
определение разделов, 232  
представления объединений и, 71  
храняемые функции и, 261  
DSS (Decision Support Systems), 313  
SQL и, 350  
запросы, сравнение с SQL, 313

## E

EXISTS, оператор  
    связанные подзапросы, 99  
    сравнение с DISTINCT, 339  
    сравнение с IN, 340  
EXPLAIN PLAN, групповые операции,  
    269

## F

FROM, инструкция  
    внешние объединения, 52  
    ограничения, 53  
    внутренние объединения, 47  
        декартово произведение, 48  
    объединения по равенству и объ-  
        единения не по равенству, 49  
    условия, 48  
    объединения, 46  
    самообъединения, 59  
        внешние, 60  
        не по равенству, 63  
FROM\_TZ, функция, 169  
FULL, ключевое слово  
    ANSI синтаксис объединения, 74

## G

GMT (Greenwich Mean Time), см. UTC,  
    153  
GROUP BY, инструкция, 82, 87, 88, 266  
    CUBE, частичная операция, 281  
    GROUPING SETS, ключевое слово,  
        288



NULL, значение, 87

ROLLUP, ключевое слово, 274, 282

ROLLUP, частичная операция, 276

UNION, операции, 268

каскадные группировки, 293

ошибки, 83

повторение названий столбцов, 290

составные столбцы, 291

фильтры, 88

GROUP\_ID, функция, 306

обзор, 299

GROUPING, функция, 288

GROUPING SETS, ключевое слово, 288

ROLLUP и CUBE как аргументы,  
297

каскадные группировки, 297

GROUPING\_ID, функция, 304

обзор, 299

## H

HAVING, инструкция, 90

ошибки, 89

скалярные подзапросы, 94

сравнение с инструкцией WHERE,  
341

## I

if-then-else, функциональность, 211

IN, оператор, 39

внешние объединения, 54

многострочные подзапросы, 96

сравнение с EXISTS, 340

INSERT, оператор, 30

в DML, 19

представления объединений, 67

преобразование строк в даты, 123

указание разделов, 233

INTERSECT, оператор работы с

множествами, 174, 177

INTERVAL DAY TO SECOND, тип

данных, 159

INTERVAL YEAR TO MONTH, тип

данных, 158

ISO, стандарт, даты

годы, 137

недели, 136

обзор, 135

## J

JOIN, ключевое слово, 71

## L

LAST\_DAY, функция, 142

LEFT, ключевое слово

ANSI-синтаксис объединения, 74

LEVEL, псевдостолбец

иерархические запросы, 197

LIKE, оператор, 41

LOCALTIMESTAMP, функция, 166

## M

MAX, функция, 79

MINUS, оператор работы с

множествами, 174, 177

сравнение таблиц, 179

MONTHS\_BETWEEN, функция, 141

## N

NEW\_TIME, функция, 147

NEXT\_DAY, функция, 143

NOT BETWEEN, оператор, 40

NOT IN, оператор, 39

многострочные подзапросы, 97

NOT, оператор

в инструкции WHERE, 37

совпадение с образцом и, 41

NTILE, аналитическая функция, 320

NULL, выражение, 42, 44

NULL, значения, 87

аналитические ранжирующие  
функции, 317

инструкция GROUP BY и, 87

обобщающие функции, 80

проверка на, 209

составные запросы, 181

функция NVL, сравнение с

функцией GROUPING, 283

NUMTODSINTERVAL, функция, 170

NUMTOYMINTERVAL, функция, 169

NVL, функция, 43

NULL-значения, сравнение с

функцией GROUPING, 283

синтаксис, 207, 209

средние значения, 81

NVL2, функция, синтаксис, 207, 209

## O

OR, оператор

в инструкции WHERE, 37

внешние объединения, 55



соображения эффективности, 45  
ORA-00904, ошибка, 114  
ORA-00932, ошибка, 209  
ORA-01402, ошибка, 114  
ORA-01427, ошибка, 96  
ORA-01468, ошибка, 53  
ORA-01476, ошибка, 222  
ORA-01733, ошибка, 69  
ORA-01779, ошибка, 113  
ORA-01790, ошибка, 182  
ORA-01861, ошибка, 125  
Oracle Supplied Packages (PL/SQL), 250  
Oracle9i, ANSI-синтаксис  
объединения, 71  
преимущества, 76  
ORDER BY, инструкция, 28  
аналитические ранжирующие  
функции, 316  
вызов хранимых функций из, 254  
иерархические запросы, 205  
операции над множествами, 184  
OUTER, ключевое слово  
ANSI-синтаксис объединения, 74

## P

PARTITION BY, инструкция  
аналитические ранжирующие  
функции, 317  
PARTITION, инструкция, 233  
PERCENT\_RANK, аналитическая  
функция, 324  
PL/SQL  
CASE-выражения и, 212  
включение SQL, 263  
обзор, 249  
переменные, преобразование в тип  
данных DATE, 123  
сводные таблицы дат, 149  
хранимые функции  
сравнение с хранимыми  
процедурами, 250  
P.M., индикатор (формат времени), 131  
PRIOR, оператор  
иерархические запросы, 195

## R

RANK, аналитическая функция, 118,  
320  
RATIO\_TO\_REPORT, аналитическая  
функция, 336

RESTRICT\_REFERENCES, псевдоком-  
ментарий, 258  
RIGHT, ключевое слово  
ANSI-синтаксис объединения, 74  
ROLLUP, ключевое слово  
групповые операции, 274, 282  
частичная операция, 276  
ROUND, функция, 144  
ROWID, псевдостолбец, 26  
ROWNUM  
инструкция GROUP BY и, 84  
ROW\_NUMBER, аналитическая  
функция, 320  
ROWS BETWEEN, инструкция  
неограниченные окна, 328  
RR (год), индикатор, 133  
RRRR (год), индикатор, 134  
RTRIM, функция, даты, 151

## S

SELECT, оператор, 26  
DISTINCT, ключевое слово, 29  
ORDER BY, инструкция, 28  
UNION, запросы, 58  
WHERE, инструкция, 25  
в DML, 20  
аналитические функции, 313  
встроенные представления, 101  
выполнение, 103  
обзор, 102  
преодоление ограничений на  
иерархические запросы, 108  
обобщающие запросы, 110  
создание наборов данных, 108  
вызов хранимых функций из, 254  
даты, диапазоны, 148  
несвязанные подзапросы  
много столбцовые, 97  
много строчные, 97  
обзор, 92  
скаляр, 93  
объединения, подзапросы, 64  
объектные типы, 241  
подзапросы, 91  
связанные подзапросы, 101  
элементы инструкции, 28  
SELF, ключевое слово  
объектные типы, 238  
SESSIONTIMEZONE, функция, 165  
SET, инструкция  
много столбцовые подзапросы, 98

SET TIME\_ZONE, инструкция, 153  
SQL (Structured Query Language), 19  
Oracle и соответствие ANSI, 21  
запросы, сравнение с запросами  
DSS, 313  
история, 21  
обзор, 19  
START WITH, инструкция, 201  
иерархические запросы, 109  
START WITH...CONNECT BY,  
инструкция, иерархические  
запросы, 195  
SUBPARTITION, инструкция, 233  
SYSTIMESTAMP, функция, 165

## T

TABLE, выражение  
доступ к коллекциям, 247  
TIMESTAMP, тип данных, 155  
TIMESTAMP WITH LOCAL TIME  
ZONE, тип данных, 157  
TIMESTAMP WITH TIME ZONE, тип  
данных, 156  
TO\_CHAR, функция, 122  
использование с функцией  
TO\_DATE, 127  
обзор, 126  
TO\_DATE, функция, 122  
использование с функцией  
TO\_CHAR, 127  
обзор, 123  
указание формата даты, 124  
формат даты по умолчанию, 123  
TO\_DSINTERVAL, функция, 171  
TO\_TIMESTAMP, функция, 166  
TO\_TIMESTAMP\_TZ, функция, 167  
TO\_YMINTERVAL, функция, 170  
TRUNC, функция, 144  
диапазоны дат и, 149  
сводные таблицы дат, 149  
TRUST, ключевое слово  
храняемые процедуры, 259  
TZ\_OFFSET, функция, 171

## U

UNION ALL, оператор работы с  
множествами, 174, 175  
сравнение таблиц, 179  
UNION, запросы, 268  
ANSI-синтаксис объединения и, 76

полные внешние объединения, 58  
UNION, инструкция, создание  
специальных наборов данных, 105  
UNION, оператор работы с  
множествами, 174, 176  
UNION, операция, сравнение с  
UNION ALL, 343  
UPDATE, оператор, 31  
CASE, выражение, 217  
DECODE, функция, 217  
в DML, 20  
встроенные представления, 112  
выборочное обобщение, 220, 222  
инструкция WHERE и, 35  
коллекции и, 248  
много столбцовые подзапросы, 98  
необязательные обновления, 218  
представления объединений, 69  
USING, инструкция объединения, 72  
UTC (Universal Coordinated Time), 153

## V

VALUE, функция  
возвращение объектов, 240  
VARCHAR2, тип данных  
функция TO\_DATE, 123

## W

WHERE, инструкция, 88  
GROUP BY, инструкция, 88  
HAVING инструкция и, 90  
сравнение WHERE и HAVING,  
341  
NULL, выражение, 42, 44  
SELECT оператор и, 25  
UPDATE оператор и, 35  
возможности, 34  
вычисление, 38  
условия, 34  
значение, 33  
логические операторы, 36  
несвязанные подзапросы, 93  
новый синтаксис объединения и, 71  
оператор внешнего объединения (+),  
51  
отсечение разделов, 234  
подзапросы, 92  
советы по использованию, 44  
столбцы, ограничение доступа, 114



условия, 34

диапазон значений, 40

компоненты, 38

принадлежность, 39

равенство/неравенство, 39

совпадение, 41

условия объединения, 44, 48

фильтрация иерархических

запросов, 199

WIDTH\_BUCKET, аналитическая

функция, 321

WITH CHECK OPTION

сокрытие столбцов, 114

WITH CHECK OPTION, инструкция

представления объединений, 68

WW (ISO неделя), индикатор, 136

## Y

YY (год), индикатор, 133

## A

аналитические функции, 311

CUME\_DIST, 324

LAG, 331

LEAD, 331

NTILE, 320

PERCENT\_RANK, 324

WIDTH\_BUCKET, 321

гипотетические, 326

для создания отчетов, 335

RATIO\_TO\_REPORT, 336

разбиение отчетов, 335

обобщающие

FIRST\_VALUE, 331

LAST\_VALUE, 331

оконные, 330

ранжирующие

DENSE\_RANK, 320

RANK, 320

ROW\_NUMBER, 320

обзор, 313

антиобъединение, 97

слиянием, 97

хешированием, 97

аргументы, ключевое слово

GROUPING SETS, 297

атрибуты, объектные типы, 239

## B

вертикальные объединения, 174

вложенные таблицы, 244

внешние ключи

ограничения, иерархическая

информация, 189

отношения и, 23

условия объединения, 48

внешние объединения, 46, 52

ограничения, 53

ошибки, 53

самообъединения, 60

синтаксис ANSI, 76

внутренние объединения, 47, 51

декартово произведение, 48

объединения по равенству и

не по равенству, 49

условия, 48

время

A.M./P.M., индикаторы, 131

доли секунды

TIMESTAMP, тип данных, 155

TIMESTAMP WITH LOCAL

TIME ZONE, тип данных, 157

TIMESTAMP WITH TIME ZONE,

тип данных, 156

обзор, 154

округление и усечение дат, 144

функции, 172

встроенные представления, 101

внешние объединения, 54

выборочное обобщение, 222

выполнение, 103

имитация аналитических запросов,  
309

обзор, 102

операторы DML, 112

ошибки, 113, 114

преодоление ограничений на

иерархические запросы, 108

обобщающие запросы, 110

создание наборов данных, 108

столбцы, сокрытие при помощи

WITH CHECK OPTION, 114

встроенные функции

совпадение с образцом, 41

вызов хранимых функций, 254

ограничения, 257



## выражения

## CASE

- UPDATE, оператор, 217
- выборочное выполнение функций, 216
- выборочное обобщение, 220, 222
- необязательные обновления, 218
- ошибки деления на ноль, 222
- преимущества, 211
- преобразование результирующих множеств, 215
- простое, 213
- с поиском, 212
- управление состоянием, 224

DECODE, функция, 208

GROUP BY, инструкция, 82, 87

NULL, значения, 42, 44

- проверка на, 209

TABLE, доступ к коллекциям, 247

обобщающие функции, 78

- ALL, ключевое слово, 81

- DISTINCT, ключевое слово, 81

- NULL, 80

- условия, 38

выходные дни, математические операции с датами и, 150

вычисление, инструкция WHERE, 38

- условия, 34

вычитание, даты, 140

## Г

гипотетические аналитические функции, 326

глобальные индексы, 228

годы

- A.D./B.C., индикаторы, 130

- ISO, стандарт, 137

- вычисление количества годов между двумя датами, 142

- двузначный формат, 133

групповые операции, 78

EXPLAIN PLAN, 269

GROUP BY, инструкция, 82, 87, 266

- CUBE, частичная операция, 281

- NULL, значение, 87

- ROLLUP, ключевое слово, 274, 282

- ROLLUP, частичная операция, 276

- WHERE, инструкция, 88

каскадные группировки, 293

повторение названий столбцов, 290

- составные столбцы, 291

GROUP\_ID, функция, 306

GROUPING, функция, 288

GROUPING SETS

- ROLLUP и CUBE как аргументы, 297

- каскадные группировки, 297

GROUPING\_ID, функция, 299, 304

HAVING, инструкция, 90

UNION, запросы, 268

обобщающие функции, 78

- NULL, 80

- ALL, ключевое слово, 81

- DISTINCT, ключевое слово, 81

суммарная информация

- GROUPING SETS, ключевое слово, 288

## Д

даты

ISO, стандарт

- годы, 137

- недели, 136

- обзор, 135

RTRIM, функция, 151

арифметические операции

- вычитание, 140

- обзор, 137

- сложение, 138

годы

- A.D./B.C., индикаторы, 130

- двузначный формат, 133

диапазоны, оператор SELECT, 148

интервальные данные, 158

- INTERVAL DAY TO SECOND,

- тип данных, 159

- INTERVAL YEAR TO MONTH,

- тип данных, 158

коды форматов, 128, 130

литералы, 135

округление/усечение, 147

подведение итогов, 151

рабочие дни, вычисления, 150

сводные таблицы, создание, 149

форматирование, 127

чувствительность к регистру, 132

функции, 172

часовые пояса

база данных, 153

обзор, 153

сеанс, 154

двузначный формат года, 133

декартово произведение, 34, 51

внутренние объединения, 48

диапазоны

даты, 148

значений, 40

уничтожение пробелов, 106

дни (рабочие), вычисления, 150

доли секунды

TIMESTAMP, тип данных, 155

TIMESTAMP WITH LOCAL TIME

ZONE, тип данных, 157

TIMESTAMP WITH TIME ZONE,

тип данных, 156

обзор, 154

дочерние узлы

иерархические запросы, 189

## З

записи, вывод в иерархическом

порядке, 201

запросы о (аналитические ранжирующие функции)

первом/последнем, 318

n лучших/худших, 318

## И

иерархические данные, представление, 190

иерархические деревья, обход, 194

иерархические запросы, 201

LEVEL, псевдостолбец, 197

PRIOR, оператор, 195

START WITH, инструкция, 201

START WITH...CONNECT BY,

инструкция, 195

записи, вывод в иерархическом

порядке, 201

концевые узлы, поиск, 192

корневые узлы

вывод, 203

поиск, 190

обобщение иерархий, 204

объединения, 205

ограничения, 205

поддеревья, удаление, 202

представления, 205

преодоление ограничений на, 108

проверка подчиненности, 201

терминология, 189

узлы, поиск родительского, 191

уровни

вывод, 203

вычисление количества, 198

фильтрация, 199

извлечение данных

таблица CUSTOMER, 24

индексы

на основе функций, 149

разделение, 228

интервальные данные (дат и времени), 158

INTERVAL DAY TO SECOND, тип

данных, 159

INTERVAL YEAR TO MONTH, тип

данных, 158

итоговая информация, 289

суммирование по датам/времени,

151

## К

каскадные группировки, 293

GROUPING SETS, 297

ключевые слова

ALL

многострочные подзапросы, 94

обобщающие функции, 79

ANY, многострочные подзапросы,

94

CUBE, частичная операция, 281

CURRENT ROW, оконные функ-

ции, 329

DBTIMEZONE, 154

DISTINCT, сравнение с оператором

EXISTS, 339

FULL, ANSI-синтаксис объедине-

ния, 74

GROUPING SETS, 288

ROLLUP и CUBE как аргументы, 297

JOIN, 71

LEFT, ANSI-синтаксис объедине-

ния, 74

OUTER, ANSI-синтаксис объедине-

ния, 74

RIGHT, ANSI-синтаксис объедине-

ния, 74



**ROLLUP**

групповые операции, 274, 282  
частичная операция, 276

**SELF**, объектные типы, 238

**TRUST**, хранимые функции, 259  
операторы работы с множествами,  
174

**ключи**

внешние

отношения и, 23  
условия объединения, 48

первичные

сравнение таблиц, 181  
условия объединения, 49

раздела, 228

разделения

оптимизатор и, 235  
хеш-разделение, 230  
таблицы, сохраняющие ключи,  
представления объединений, 65

**коллекции**

изменение, 248

обращение к, 247

композиционное разделение, 230

**константы**

инструкция **GROUP BY**, 84

**концевые узлы**

иерархические запросы, 190  
поиск, 192

**корневые узлы**

вывод, 203  
иерархические запросы, 190  
поиск, 190

**Л**

литералы, даты, 135

логические операторы

**WHERE**, инструкция, 36

локальные индексы, 228

**М**

математические операции с датами

вычитание, 140

обзор, 137

сложение, 138

**месяцы**

математические операции с датами,  
139

получение первого дня, 143

получение последнего дня, 142

**минуты**

математические операции с датами,  
139

многостолбцовые подзапросы, 97

многострочные подзапросы, 97

ошибки, 96

множества, операции над, 173

**INTERSECT**, оператор, 174, 177

**MINUS**, оператор, 174, 177

сравнение таблиц, 179

**UNION**, оператор, 174, 176

**UNION ALL**, оператор, 174, 175

сравнение таблиц, 179

ограничения, 186

модель объект-отношение

тестовая база данных, 22

**Н****недели**

**ISO**, стандарт, 136

математические операции с датами,  
138

непроцедурные языки, 19

несвязанные подзапросы, 92

многостолбцовые, 97

многострочные, 97

обзор, 92

скаляр, 93

неявные преобразования типов

**DATE**, тип данных, 124

**О**

обобщающие аналитические функции

**FIRST\_VALUE**, 331

**LAST\_VALUE**, 331

обобщающие запросы

преодоление ограничений, 110

обобщающие функции, 78, 289

**ALL**, ключевое слово, 81

**DISTINCT**, ключевое слово, 81

**GROUP BY** инструкция и, 83

**NULL**, 80

ошибки, 82

обобщение иерархий, 204

обход, иерархические деревья, 194

объединения, 46

**ANSI-синтаксис**, 71

преимущества, 76

**USING**, инструкция, 72

антиобъединения, 97



- вертикальные, 174
- внешние, 52
  - ограничения, 53
  - полные, 55, 58
  - синтаксис ANSI, 76
- внутренние, 47
  - декартово произведение, 48
  - объединения по равенству и не по равенству, 49
  - условия, 48
- иерархические запросы, 205
- имитация аналитических запросов, 309
- подзапросы, 64
- представления, таблицы, сохраняющие ключи, 65
- самообъединения, 59
  - внешние, 60
  - не по равенству, 63
- сравнение объединений по равенству и не по равенству, 49
- условия
  - WHERE, инструкция, 25
  - новый синтаксис и, 73
- хранимые функции, проблемы, 256
- частичные объединения, 100
- объектные типы, 237
  - атрибуты, 239
  - параметры, 241
  - таблицы, 239
- объекты, 22
- оконные аналитические функции, 330
- округление дат, 147
- оператор внешнего объединения (+), 51
  - внешнее самообъединение, 60
- оператор проверки
  - неравенства ( $\neq$ ), 39
  - равенства ( $=$ ), 39
- операторы
  - AND, 36
  - BETWEEN, 40
  - CREATE TABLE
    - объектные таблицы, 239
  - CREATE TYPE BODY, 238
  - DELETE, 31
    - много столбцовые подзапросы, 98
    - представления объединений, 68
    - скалярные подзапросы, 94
  - EXISTS
    - связанные подзапросы, 99
    - сравнение с ключевым словом DISTINCT, 339
    - сравнение с оператором IN, 340
  - IN, 39
    - внешние объединения, 54
    - много строчные подзапросы, 96
  - INSERT, 30
    - представления объединений, 67
    - преобразование строк в даты, 123
    - указание разделов, 233
  - LIKE, 41
  - NOT, 37
    - совпадение с образцом и, 41
  - NOT BETWEEN, 40
  - NOT IN, 39
    - много строчные подзапросы, 97
  - OR, 37
    - внешние объединения, 55
    - соображения эффективности, 45
  - PRIOR, иерархические запросы, 195
  - SELECT, 26
    - DISTINCT, ключевое слово, 29
    - ORDER BY, инструкция, 28
    - UNION, операции, 58
    - WHERE, инструкция, 25
    - встроенные представления, 101, 102, 103, 108, 110
    - выбор данных по диапазону дат, 148
    - несвязанные подзапросы, 92
      - много столбцовые, 97
      - много строчные, 97
      - скаляр, 93
    - объектные типы, 241
    - подзапросы, 91
    - связанные подзапросы, 101
    - элементы инструкции, 28
  - UPDATE, 31
    - CASE, выражение, 217
    - DECODE, функция, 217
    - встроенные представления, 112
    - выборочное обобщение, 220, 222
    - много столбцовые подзапросы, 98
    - необязательные обновления, 218
    - представления объединений, 69
  - больше или равно ( $\geq$ ), 41
  - больше чем ( $>$ ), 41
    - самообъединения не по равенству, 63
  - в DML, 19

- внешнее объединение (+), 51
  - внешние самообъединения, 60
- вычитание (-)
  - даты, 140
- логические, инструкция WHERE, 36
- меньше или равно ( $\leq$ ), 41
- меньше чем ( $<$ ), 41
  - самообъединения по равенству, 63
- неравенство ( $\neq$ ), 39
- приоритет, 45
- работа с множествами, 174
  - INTERSECT, 174, 177
  - MINUS, 174, 177, 179
  - UNION, 174, 176
  - UNION ALL, 174, 175, 179
- равенство ( $=$ ), 39
- скалярные подзапросы, 93
- сравнения
  - многострочные подзапросы, 94
  - неравенство ( $\neq$ ), 39
  - подзапросы, 40
  - равенство ( $=$ ), 39
  - условие объединения, 49
  - условия, 38
  - хранимые функции и, 261
- операции над множествами
  - сравнение таблиц, 181
- оптимизатор
  - антиобъединения, 97
  - вычисление условий, 35
  - ключи разделения и, 235
  - отсечение разделов, 234
  - указание разделов, 233
  - частичные объединения, 100
- отношения, 23
  - представление иерархических данных, 190
  - рекурсивные, обход, 108
  - связанные подзапросы, 99
  - таблицы, сохраняющие ключи, 67
  - типы коллекций, 243
    - вложенные таблицы, 244
    - переменные массивы, 243
    - разборка, 246
    - создание, 245
- отсечение разделов, 237
- ошибки
  - DECODE, функция, 209
  - GROUP BY, инструкция, 83
  - HAVING, инструкция, 89
  - внешние объединения, 53
  - встроенные представления, 113–114
  - деление на ноль, 222
  - добавление столбцов, 114
  - многострочные подзапросы, 96
  - обобщающие функции, 82
  - представления объединений, 69
  - преобразования типа данных
    - DATE, 125
  - составные запросы, 182
- П**
  - пакеты
    - спецификация и тело, 251
    - хранимые процедуры и функции, 250
  - параметры, объектные типы, 241
  - первичные ключи
    - сравнение таблиц, 181
    - условия объединения, 49
  - перегрузка функций, 253
  - переменные
    - массивы, 243
    - связанные, 346
  - поддеревья, удаление, 202
  - подзапросы, 35, 91
    - внешние объединения, 55
    - встроенные представления, 101
    - выполнение, 103
    - обзор, 102
    - преодоление ограничений на иерархические запросы, 108
    - обобщающие запросы, 110
    - создание наборов данных, 108
  - несвязанные
    - многостолбцовые, 97
    - многострочные, 97
    - обзор, 92
    - скаляр, 93
  - операторы сравнения и, 40
  - разбор примера, 120
  - связанные, 101
  - подразделы, 231
    - удаление, 233
  - полные внешние объединения, 55, 58
  - представления
    - внешние объединения, 54
    - встроенные, 101
    - выборочное обобщение, 222
    - выполнение, 103



- обзор, 102
- операторы DML, 112
- преодоление ограничений на
  - иерархические запросы, 108
  - обобщающие запросы, 110
- создание наборов данных, 108
- сокрытие столбцов при помощи
  - WITH CHECK OPTION, 114
- иерархические запросы, 205
- объединений
  - DELETE, оператор, 68
  - INSERT, оператор, 67
  - UPDATE, оператор, 69
  - операторы DML и, 71
  - ошибки, 69
- разделы, 234
- словарь данных
  - USER\_UPDATABLE\_COLUMNS, 70
- хранимые функции, 255
- приоритет операторов/условий, 45
- производительность
  - OR, оператор, соображения
    - эффективности, 45
  - выборочное выполнение функции, 216
- разделение и, 227
- простые CASE-выражения, 213
- процедуры хранимые, сравнение с
  - хранимыми функциями, 250
- псевдонимы
  - таблиц, 351
  - SELECT, оператор, 26
  - самообъединения, 59
  - случаи использования, 47
- новый синтаксис объединения и, 72
- столбцы, встроенные представления, 102
- псевдостолбцы ROWID, 26

## Р

- рабочие дни, вычисления, 150
- разбор излишний, 345
- разборка коллекций, 246
- разделение
  - индексы, 228
  - ключи, 228
  - комPOSITное, 230
  - по диапазону, 229
  - по списку, 232

- хеш, 230
- разделы
  - определение, 232
  - отсечение, 237
  - представления, 234
  - таблицы, обзор, 226
  - хранение, 227
- ранжирующие аналитические функции
  - DENSE\_RANK, 320
  - RANK, 320
  - ROW\_NUMBER, 320
  - обзор, 313
- результатирующие множества, 24
  - WHERE, инструкция, условия, 34
  - декартово произведение, 34, 48
  - ограничение возвращаемого набора
    - WHERE, инструкция, 25
  - операции над множествами,
    - названия столбцов, 183
  - подзапросы, 92
  - поиск данных, отсутствующих в
    - базе данных, 106
  - преобразование
    - CASE, выражение, 215
    - DECODE, функция, 214
- рекурсивные отношения, обход, 108
- родительские узлы
  - иерархические запросы, 189
  - поиск, 191

## С

- самообъединения, 46, 59
  - внешние, 60
  - не по равенству, 63
- сводки, инструкция GROUP BY, 82
- сводные таблицы дат, создание, 149
- связанные
  - переменные, 346
  - подзапросы, 92, 101
- сеанс, часовые пояса, 154
- секунды, доли
  - TIMESTAMP, тип данных, 155
  - TIMESTAMP WITH LOCAL TIME
    - ZONE, тип данных, 157
  - TIMESTAMP WITH TIME ZONE,
    - тип данных, 156
  - обзор, 154
- символы
  - преобразование в даты, 125
  - совпадение с образцом, 41



- скалярные подзапросы, 93
- скобки ( )
  - подзапросы, 91
  - приоритет операторов/условий, 45
- словарь данных, представление
  - USER\_UPDATABLE\_COLUMNS, 70
- сложение, даты, 138
- совместимость по операции слияния
  - условия, 174
- совпадение с образцом
  - встроенные функции, 41
  - условия, 41
- сортировка результатов запроса
  - ORDER BY, инструкция, 28
- составные запросы, 173
  - NULL, значение, 181
  - ошибки, 182
- спецификация пакета, 251
- средние значения
  - обобщающие функции, 81
- стандарты ANSI, 21
- столбцы
  - GROUP BY, инструкция
    - каскадные группировки, 293
  - GROUPING SETS, каскадные группировки, 297
  - LEVEL, псевдостолбец, 197
  - названия
    - операции над множествами, 183
    - псевдонимы таблиц и, 47
  - обновляемые
    - просмотр всех в схеме, 70
  - повторение названий
    - GROUP BY, инструкция, 290
  - псевдонимы
    - встроенные представления, 102
  - разделение по диапазону, 229
  - сокрытие, WITH CHECK OPTION, 114
  - составные
    - инструкция GROUP BY, 291
- строки
  - преобразование в даты, 123, 125
  - совпадение с образцом, 41
  - создание из дат, 126
- сценарии PL/SQL, 250

## T

- таблицы, 52
  - ANSI-синтаксис объединения, 71
  - внешние объединения, 76
  - преимущества, 76

## CUSTOMER

- пример оператора SELECT, 24
- внешние объединения, 53
  - ограничения, 53
  - полные, 55, 58
- внутренние объединения, 47
- изменение
  - без инструкции WHERE, 33
  - оператор UPDATE, 35
- объединения
  - по равенству и не по равенству, 49
  - подзапросы, 64
- объектный тип, 239
- объекты, 22
- псевдонимы, 351
  - случаи использования, 47
- разделение, обзор, 226
- самообъединения, 59
  - внешние, 60
  - не по равенству, 63
- сохраняющие ключи
  - представления объединений, 65
- сравнение с помощью операций над множествами, 181
- столбцы NOT NULL и оператор INSERT, 30
- тело пакета, 251
- терминология, 22
  - иерархические запросы, 189
- тестовая база данных
  - модель объект-отношение, 22
  - обзор, 22
- типы данных
  - DATE
    - NULL-значения и, 182
    - внутренний формат хранения, 122
    - преобразование, 122
    - указание формата, 124
    - формат по умолчанию, 123
  - объектные типы, 237
    - атрибуты, 239
    - параметры, 241
    - таблицы, 239
    - условия объединения, 49
- типы коллекций
  - отношения, 243
    - вложенные таблицы, 244
    - переменные массивы, 243
  - разборка, 246

создание, 245

## У

удаление

- поддеревьев (иерархические запросы), 202
- подразделов, 233

узлы

- иерархические запросы, 189
- концевые, поиск, 192
- корневые
  - вывод, 203
  - поиск, 190
- поиск родительского узла, 191

уровни

- вывод, 203
- вычисление количества, 198
- иерархические запросы, 190
- чистоты
  - хранимые функции, 258

усечение дат, 147

условия

- CASE, выражение, 212
- в инструкции WHERE, 34
- внутренние
  - объединения по равенству и не по равенству, 49
- внутренние объединения, 48
- диапазон значений, 40
- компоненты, 38
- неравенства, 39
- объединения
  - WHERE, инструкция, 44
  - новый синтаксис и, 73
- принадлежности, 39
- приоритет, 45
- равенства, 39
- совместимость по операции
  - слияния, 174
- совпадения, 41

условная логика, примеры

- использования CASE и DECODE, 225

## Ф

фильтры

- HAVING, инструкция, 88
- иерархические запросы, 199

формат даты по умолчанию, 123

форматирование даты, 127

- округление и усечение, 145
- стандарт ISO, 135

текстовые компоненты, 132

чувствительность к регистру, 132

функции

ADD\_MONTHS, 139

- вычитание дат, 141

CAST, 169

CURRENT\_DATE, 165

CURRENT\_TIMESTAMP, 165

DBTIMEZONE, 164

DECODE

- UPDATE, операторы, 217

- выборочное выполнение

  - функции, 216

- выборочное обобщение, 220, 222

- необязательные обновления, 218

- ошибки деления на ноль, 222

- преобразование результирующих множеств, 214

- синтаксис, 207, 208

- управление состоянием, 223

FROM\_TZ, 169

GROUP\_ID, 306

- обзор, 299

GROUPING, 288

GROUPING\_ID, 304

- обзор, 299

LAST\_DAY, 142

LOCALTIMESTAMP, 166

MAX, 79

MONTHS\_BETWEEN, 141

NEW\_TIME, 147

NEXT\_DAY, 143

NUMTODSINTERVAL, 170

NUMTOYMINTERVAL, 169

NVL, 43

- синтаксис, 207, 209

- сравнение с функцией GROUPING, 283

NVL2, синтаксис, 207, 209

ROUND, 144

RTRIM, даты, 151

SESSIONTIMEZONE, 165

SYSTIMESTAMP, 165

TO\_CHAR, 122

- обзор, 126

TO\_DATE, 122

- обзор, 123

- указание формата даты, 124

- формат даты по умолчанию, 123

TO\_DSINTERVAL, 171

TO\_TIMESTAMP, 166

TO\_TIMESTAMP\_TZ, 167

TO\_YMINTERVAL, 170

функции

TRUNC, 144

TZ\_OFFSET, 171

VALUE, возвращение объектов, 240

аналитические

CUME\_DIST, 324

LAG, 331

LEAD, 331

NTILE, 320

PERCENT\_RANK, 324

WIDTH\_BUCKET, 321

гипотетические, 326

обобщающие

FIRST\_VALUE, 331

LAST\_VALUE, 331

оконные, 330

ранжирующие

DENSE\_RANK, 320

RANK, 320

ROW\_NUMBER, 320

обзор, 313

создании разбиение отчетов,  
335

встроенные, совпадение с образцом,  
41

математические для дат, 138

обобщающие, 78

ALL, ключевое слово, 81

DISTINCT, ключевое слово, 81

NULL, 80

обработка дат и времени, 172

перегрузка, 253

работа с временными интервалами,  
172

хранимые функции

TRUST, ключевое слово, 259

в операторах DML, 261

вызов, 254, 257

правила, 260

представления, 255

хранимые функции

проблемы при объединении, 256

сравнение с хранимыми

процедурами, 250

уровни чистоты, 258

## Х

хеш-разделение, 230

хранение разделов, 227

хранимые процедуры

пакеты, 250

сравнение с хранимыми

функциями, 250

хранимые функции

TRUST, ключевое слово, 259

в операторах DML, 261

вызов, 254

ограничения, 257

пакеты, 250

правила, 260

представления, 255

проблемы при объединении, 256

уровни чистоты, 258

## Ч

часовые пояса

NEW\_TIME, функция, 147

база данных, 153

обзор, 153

сеанс, 154

частичные объединения, 100

слиянием, 100

хешированием, 100

числа, преобразование в даты, 123, 125

## Я

языки программирования

непроцедурные языки, 19